

JS Level 1



Введение



Работа с HTTP

В этой лекции мы с вами поговорим о загрузке данных по протоколу HTTP и об общении с сервером в целом. Кроме того, закроем тему с событиями, поговорив об Event Listener'ах.

В качестве основы мы возьмём идею с прошлой лекции, но немного её видоизменим, чтобы было проще.

Для работы вам потребуется ещё один сервер – помимо Live Server'а. Вы его можете скачать по адресу <https://alif-skills.pro/media/lection-server.js> (для запуска будет нужен Node.js)



Запуск сервера

Чтобы запустить сервер, откройте его в VS Code (не забудьте предварительно распаковать из zip), затем откройте терминал: **Ctrl + `**. В терминале введите команду **node lection-server.js** (\$ писать не нужно):

```
$ node lection-server.js  
server started at http://127.0.0.0.1:9999
```



Если сервер запустится успешно, в терминале появится следующее сообщение:



Запуск сервера

Важно: при дальнейшей работе сервер должен быть запущен (т.е. не надо закрывать вкладку с сервером). Вы всегда можете остановить сервер с помощью клавиатурного сокращения **Ctrl + C**.



Сервер

Сервер запускается командой `node lection-server.js`. Если вдруг, при запуске вы видите вот такую ошибку:

```
events.js:287
    throw er; // Unhandled 'error' event
    ^
```

```
Error: listen EADDRINUSE: address already in use :::9999
```

Это значит, что вы либо второй раз пытаетесь запустить сервер (а его нужно запускать только один раз), либо какая-то программа на вашем компьютере уже работает на по этому адресу (кроме того, сервер может блокировать антивирус, поэтому посмотрите в настройках антивируса).



Сервер

Чтобы узнать, какая программа уже работает по этому адресу, выполните следующие действия:

1. Вбейте в терминале команду **resmon**
2. Перейдите на вкладку Сеть и найдите ID процесса, который занимает порт 9999:

Монитор ресурсов

Файл Монитор Справка

Обзор ЦП Память Диск Сеть

Процессы с сетевой активностью

Сетевая активность 1 кбит/с - сетевой ввод-вывод Использование сети: 0%

TCP-подключения

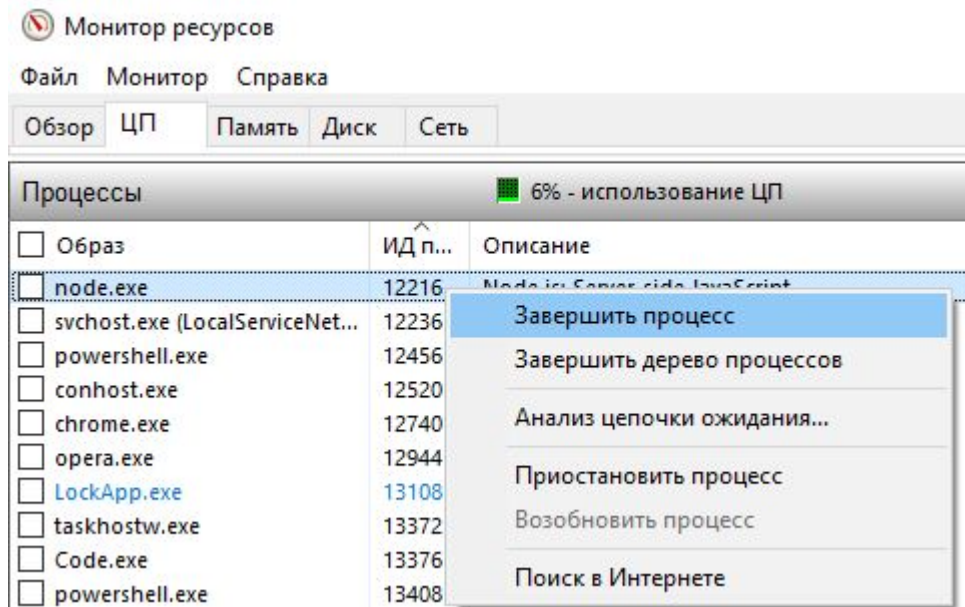
Прослушиваемые порты

Образ	ИД процесса	Адрес	Порт	Протокол	Состояние бр...
goland64.exe	19744	Петлевой адрес в IPv4	6943	TCP	Разрешен, не ...
node.exe	12216	IPv6 не задан	9999	TCP	Не разрешен, ...



Сервер

3. Перейдите на вкладку ЦП, найдите этот процесс по идентификатору, щёлкните правой кнопкой мыши нажмите Завершить процесс:



Сервер

В Mac OS выполните команду:

```
lsof -nP -iTCP:9999 | grep LISTEN
```

Вы получите идентификатор процесса примерно вот в таком виде:

```
node 62441 ...
```

После чего этот процесс можно убить командой:

```
kill -9 62441 либо sudo kill -9 62441
```

В Linux выполните команду:

```
fuser -n tcp 9999
```

Вы получите идентификатор процесса примерно вот в таком виде:

```
9999/tcp: 62441
```

После чего этот процесс можно убить командой:

```
kill -9 62441 либо sudo kill -9 62441
```



Повторение



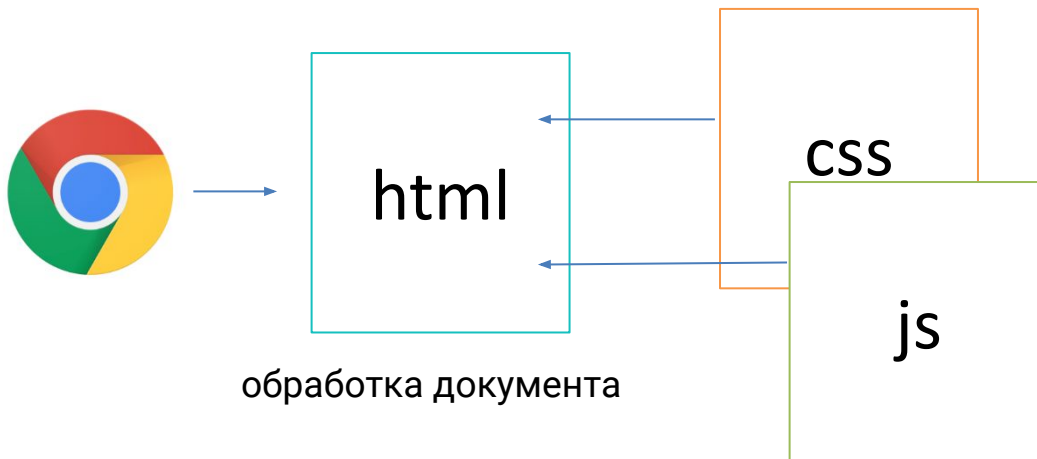
Web Application

На прошлой лекции мы поговорили с вами, как работают веб-приложения (а именно их клиентская часть) – они загружаются и запускаются в браузере:



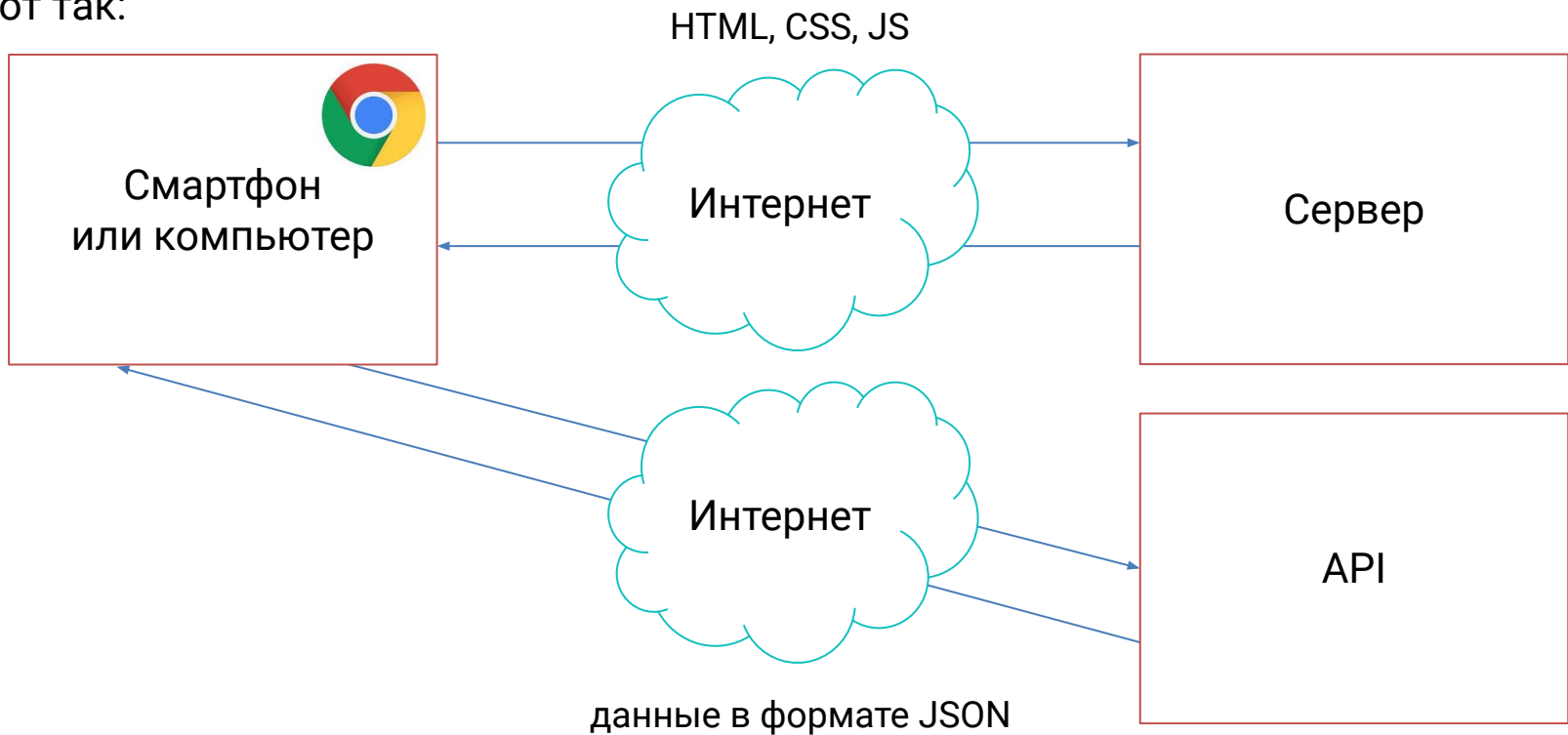
Ресурсы

Кроме того, мы обсудили сам механизм: сначала загружается HTML-документ (если вы указали его в адресной строке), а затем уже все ресурсы, которые в этом самом документе прописаны:



API

Сегодня наша задача поговорить о том, что сервер и веб-приложение могут обмениваться данными в различных форматах без перезагрузки страницы. Для этого мы возьмём ещё один сервер (назовём его API) и картинка будет выглядеть вот так:



Задача



WishList

Мы хотим, чтобы желания не просто хранились локально в нашем браузере (и исчезали после перезагрузки), а мы хотим, чтобы они хранились на сервере и даже если мы перезагружаем страницу, ничего с ними не происходило.

Так работает большая часть приложений, например, когда вы работаете с Facebook, то данные хранятся на серверах Facebook и если вы закроете браузер и снова откроете – ничего с вашими данными случиться не должно.

Для этого нам нужно понять, как это делается, и какие инструменты в JS у нас для этого есть.



WishList

За основу мы возьмём идею проекта с прошлой лекции, но сделаем так, чтобы все данные загружались с сервера и сохранялись тоже на сервер.



WishList

Приступим, наша разметка:

```
<> index.html > ...
1  <!DOCTYPE html>
2  <html lang="ru">
3
4  <head>
5      <meta charset="UTF-8">
6      <title>WishList</title>
7      <link rel="stylesheet" href="css/styles.css">
8  </head>
9
10 <body>
11     <noscript>Необходимо включить поддержку JavaScript в вашем браузере.</noscript>
12     <div id="root"></div>
13     <script src="js/app.js"></script>
14 </body>
15
16 </html>
```



WishList

Код в app.js:

```
js > JS app.js > ...  
  1  'use strict';  
  2  
  3  const rootEl = document.getElementById('root');  
  4
```



HTTP



HTTP

HTTP (HyperText Transfer Protocol) – это протокол передачи данных. Изначально создавался на то, чтобы передавать HTML-документы, но в настоящее время позволяет передавать практически любой тип контента.

Т.е. когда мы с вами рисуем вот такую картинку, то мы подразумеваем, что все данные передаются по протоколу HTTP:



Протокол

Что такое протокол? Протокол – это правила общения двух сторон. В нашем случае браузера и сервера. Вы можете воспринимать это как язык: например, если один человек говорит на английском, а другой на китайском – они вряд ли друг друга поймут.

Протокол устанавливает правила общения: говорим на таком-то языке, в таком-то формате, разрешено передавать такие-то сообщения.

Как всегда, мы будем достаточно упрощённо всё рассматривать, опуская некоторые детали, но мы вам настоятельно рекомендуем ознакомиться с теми ссылками, которые будут указаны на следующей странице.



Версии протокола

Ключевых версий на данный момент три:

1.1: <https://tools.ietf.org/html/rfc2616>

2.0: <https://tools.ietf.org/html/rfc7540>

3.0: <https://tools.ietf.org/html/draft-ietf-quic-http-34>

Мы будем с вами рассматривать версию 1.1 (поскольку она самая распространённая), но потихоньку все переходят на версию 2.0.



HTTP 1.1

Версия 1.1 является текстовой. Что это значит? Это значит, что все данные, которые передаются, можно представить в виде текста.

Например, вы можете открыть вкладку **Network**, выбрать конкретный файл и увидеть, как происходит передача:

The screenshot displays the Network tab of a web browser's developer tools. On the left, a list of resources is shown: index.html (highlighted), styles.css, app.js, and ws. The main panel on the right shows the details for the selected 'index.html' resource. It includes tabs for Headers, Preview, Response, Initiator, and Timing. The 'General' section is expanded, showing the following information:

- Request URL: `http://127.0.0.1:5500/index.html`
- Request Method: `GET`
- Status Code: `200 OK` (indicated by a green dot)
- Remote Address: `127.0.0.1:5500`
- Referrer Policy: `no-referrer-when-downgrade`

Below the General section, the 'Response Headers' are listed, with a 'view source' link next to them:

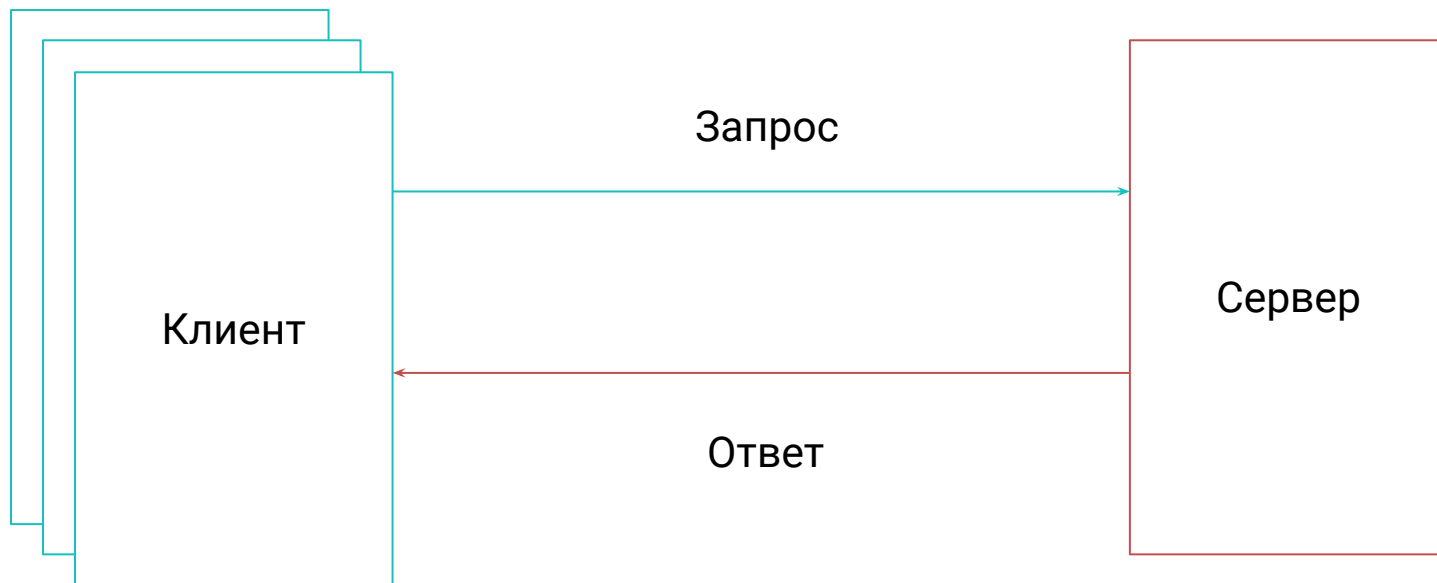
- Accept-Ranges: `bytes`
- Access-Control-Allow-Credentials: `true`
- Cache-Control: `public, max-age=0`
- Connection: `keep-alive`
- Content-Length: `1977`
- Content-Type: `text/html; charset=UTF-8`



HTTP 1.1

В рамках этого протокола общение строится в формате запрос-ответ, т.е. клиент должен что-то запросить у сервера, а сервер ему на это должен что-то ответить.

Под клиентом в данном случае понимается веб-браузер:



Естественно, клиентов может быть много. Серверов, конечно же, тоже, но мы рассмотрим простейший случай, когда сервер только один.



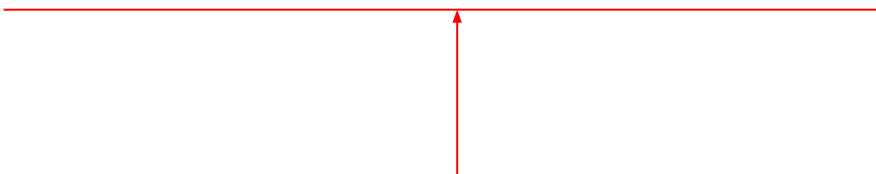
Message

И запрос, и ответ называют message (или сообщение). В рамках спецификации это выглядит вот так:

4.1 Message Types

HTTP messages consist of requests from client to server and responses from server to client.

HTTP-message = Request | Response ; HTTP/1.1 messages



Есть всего два типа сообщений: запрос (request) и ответ (response)

Символ | означает ИЛИ в рамках спецификации



Запрос

Запрос состоит из трёх частей:

5 Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

```
Request      = Request-Line           ; Section 5.1
               *(( general-header      ; Section 4.5
                 | request-header      ; Section 5.3
                 | entity-header ) CRLF) ; Section 7.1
               CRLF
               [ message-body ]       ; Section 4.3
```

1. Request Line (строка запроса) – что мы хотим получить от сервера
2. Headers (заголовки) – мета-данные
3. Message Body (тело) – что мы передаём на сервер (если, например, загружаем файл)



Запрос

Пока всё выглядит достаточно абстрактно, но если вы в той же панельке [Network](#) выделите [index.html](#) и пролистаете чуть ниже, то увидите:

▼ Request Headers [view source](#)


```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate, br
Accept-Language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7
Cache-Control: no-cache
Connection: keep-alive
Host: 127.0.0.1:5500
Pragma: no-cache
```

Нажмите на [view source](#).



Запрос

Вот так оно на самом деле отсылается (текстом, поэтому и протокол текстовый):



```
▼ Request Headers  view parsed
GET /index.html HTTP/1.1 ← request line
Host: 127.0.0.1:5500
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrom
e/81.0.4044.129 Safari/537.36 OPR/68.0.3618.63
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
```

Тело у этого конкретного запроса отсутствует. То же самое можно посмотреть и с ответом.



Ответ

Ответ состоит из трёх частей:

6 Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

```
Response      = Status-Line           ; Section 6.1
                *(( general-header     ; Section 4.5
                  | response-header    ; Section 6.2
                  | entity-header ) CRLF) ; Section 7.1
                CRLF
                [ message-body ]       ; Section 7.2
```

1. Status Line (строка запроса) – насколько успешно завершился наш запрос
2. Headers (заголовки) – мета-данные
3. Message Body (тело) – что сервер передаёт нам в ответ (например, index.html)



Данные

В принципе, это почти всё, что нам нужно знать: есть запрос и ответ, у каждого по три части.

Если частей всего три, значит данные мы можем передавать тремя разными способами (или их комбинацией):

1. В Request Line
2. В Headers
3. В Message Body

И получать тоже можем тремя разными способами (или их комбинацией):

1. В Status Line
2. В Headers
3. В Message Body



Инструменты

Остаётся лишь один вопрос: как нам из JS этот самый запрос отправить?



XMLHttpRequest



XMLHttpRequest

В рамках JS у нас есть специальный объект `XMLHttpRequest` (коротко – xhr), который позволяет осуществлять запросы из кода JS без перезагрузки страницы.

На этот объект есть своя [спецификация](#). Чаще всего вы будете встречать описание этой возможности под термином AJAX (Asynchronous JavaScript And XML), но в современном мире он уже не расшифровывается, а просто подразумевает возможность совершать HTTP-запросы без перезагрузки страницы.

Давайте посмотрим, как с ним работать.



Разметка

В рамках сегодняшней лекции мы рассмотрим подход с [innerHTML](#), заодно поговорим, какие проблемы он за собой влечёт:

```
1  'use strict';
2
3  const rootEl = document.getElementById('root');
4  rootEl.innerHTML = `
5      <h1>WishList</h1>
6      <div data-id="loader">Загружаем данные...</div>
7      <form data-id="wish-form">
8          <div data-id="message"></div>
9          <div>
10             <label for="name-input">Название</label>
11             <input data-input="name" id="name-input">
12          </div>
13          <div>
14             <label for="price-input">Цена</label>
15             <input data-input="price" id="price-input" type="number">
16          </div>
17          <div>
18             <label for="description-input">Описание</label>
19             <textarea data-input="description" id="description-input" rows="5"></textarea>
20          </div>
21          <button>Добавить</button>
22      </form>
23      <div>Необходимо: <span data-id="total">0</span> с.</div>
24      <ul data-id="wish-list"></ul>
25  `;
```



XMLHttpRequest

Использовать этот объект достаточно просто, общая схема выглядит следующим образом:

1. Создаём объект
2. Настраиваем, указывая, куда отправлять запрос и каким образом
3. Отправляем запрос
4. Получаем ответ



Создаём объект

Обычно объекту дают имя `xhr` (сокращённо от `XMLHttpRequest`):

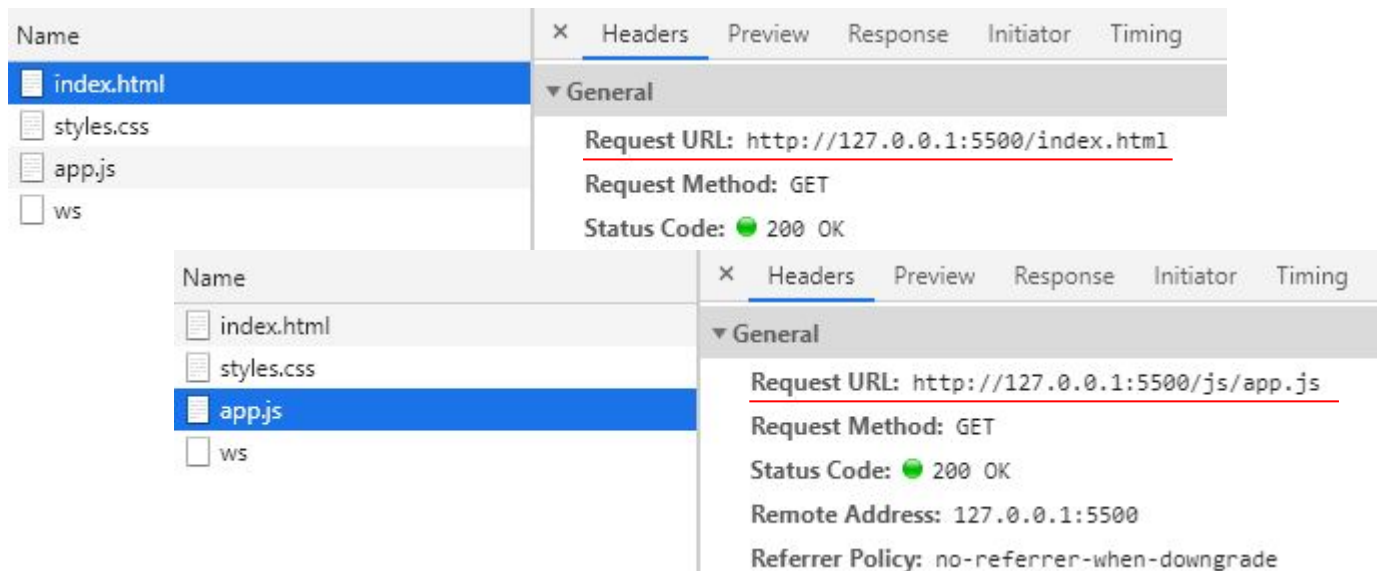
```
28  const xhr = new XMLHttpRequest();
```



Настройка

А дальше возникает вопрос: что значит "указывая куда и каким образом отправлять запрос".

В Web у нас с вами есть адреса, их называют URL. Соответственно, запросы отправляются именно на адреса:



Адреса

В качестве адреса можно указывать только путь (как мы делали, например, `src="js/app.js"`), тогда браузер сам "вычислит" полный адрес относительно того, откуда загружена вся страница.

Либо можно указывать полный адрес, например <http://127.0.0.1:9999/api/wishes>, тогда уже не важно*, откуда загружена наша страница.

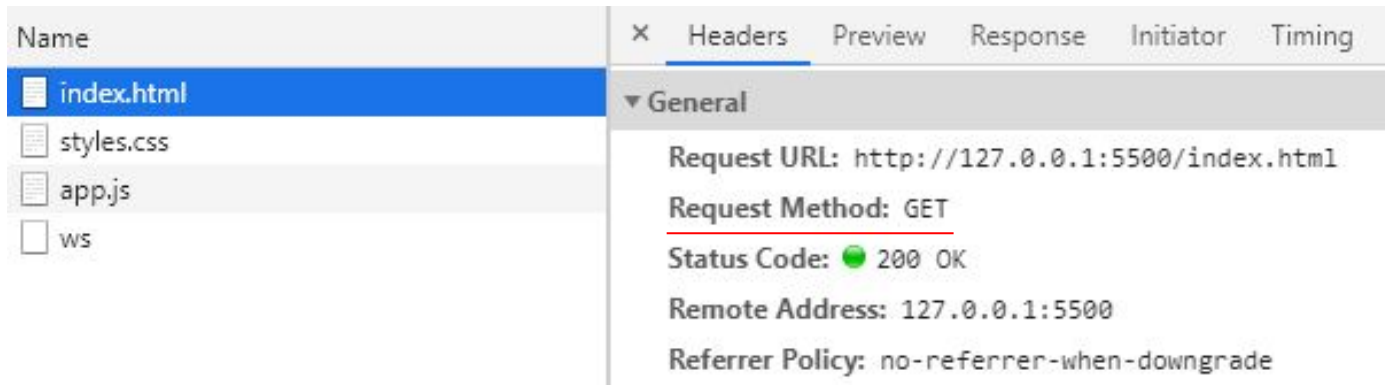
Наш сервер с данными запущен именно на <http://127.0.0.1:9999> и доступ мы хотим получить к пути `/api/wishes`, поэтому будем указывать полный адрес.

Примечание*: это не совсем так, но для нас это станет важно только на следующем курсе.



Method

Помимо URL в запросе ещё указывается метод:



Что это такое? В рамках протокола HTTP определён набор "слов", которые называются методами. У этих методов есть смысловая нагрузка, например, метод **GET** используется для получения данных, а **POST** – для отправки. Кроме того, в рамках конкретных методов есть и ограничения, например метод **GET** должен содержать пустое тело (вспоминайте, что запрос состоит из трёх частей: Request Line, Headers и Body).



Method

5.1.1 Method

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

Method	= "OPTIONS"	; Section 9.2
	"GET"	; Section 9.3
	"HEAD"	; Section 9.4
	"POST"	; Section 9.5
	"PUT"	; Section 9.6
	"DELETE"	; Section 9.7
	"TRACE"	; Section 9.8
	"CONNECT"	; Section 9.9
	extension-method	
extension-method	= token	

Как определить, как метод нужно использовать и какой адрес?



API

На самом деле, решаете это не вы, а человек, который пишет сервер (вы это будете делать на следующих курсах).

Именно он вам говорит нечто вроде:

- Для получения списка желаний отправляй GET-запрос на <http://127.0.0.1:9999/api/wishes>
- Для добавления желания отправляй POST-запрос на <http://127.0.0.1:9999/api/wishes>, в теле запроса должен быть JSON с данными
- Для удаления желания отправляй запрос DELETE на <http://127.0.0.1:9999/api/wishes/{id}>, где {id} – это идентификатор желания
- Для обновления желания отправляй ...

И т.д.



API

И это тоже называется API – потому что это определение программного взаимодействия нашего приложения и сервера.

Хорошо, давайте смотреть:

```
28  const apiUrl = 'http://127.0.0.1:9999/api/wishes';  
29  
30  const xhr = new XMLHttpRequest();  
31  xhr.open('GET', apiUrl, false); // async false
```

Несмотря на то, что метод называется `open`, на самом деле, ничего он не открывает, он всего лишь "настраивает" наш объект.

Мы маленько схитрим и укажем третьим параметром `false`. Третий параметр отвечает за асинхронность. Что это такое – мы как раз разберём в процессе. В данный момент мы отправляем не асинхронный запрос, т.е. синхронный.



Отправляем запрос

Осталось только отправить сам запрос. Для этого у нас есть метод `send`. Поскольку тело GET-запроса должно быть пустым, мы просто ничего не указываем:

```
const apiUrl = 'http://127.0.0.1:9999/api/wishes';  
  
const xhr = new XMLHttpRequest();  
xhr.open('GET', apiUrl, false);  
xhr.send();  
console.log(xhr.response);
```



Если мы отправляем синхронный запрос, то ответ нам придёт в качестве свойства `response` объекта `xhr` после завершения метода `send`.

Можете вывести в консоль `response` и удостовериться, что вам приходят данные. Мы же пока разберёмся, что значит синхронность и асинхронность.



Event Loop



Freeze

А теперь откройте страницу на новой вкладке – вы увидите белый экран. В чём же причина?

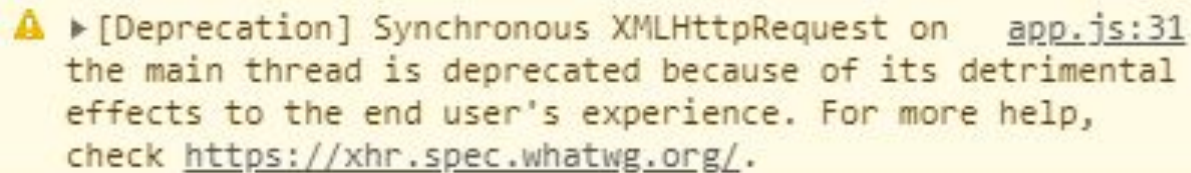
Мы специально написали медленный сервер, чтобы эмулировать "занятость" сервера. Сервера не всегда отвечают сразу + у вас может быть не очень быстрый интернет.

И на самом деле произошёл freeze. Но почему? В прошлый раз – понятно, браузер делал работу (считал бессмысленно до нескольких миллиардов), а в этот же раз он просто послал запрос.



Консоль

Посмотрим в консоль и увидим там следующее сообщение:



```
⚠ ▶ [Deprecation] Synchronous XMLHttpRequest on app.js:31  
the main thread is deprecated because of its detrimental  
effects to the end user's experience. For more help,  
check https://xhr.spec.whatwg.org/.
```

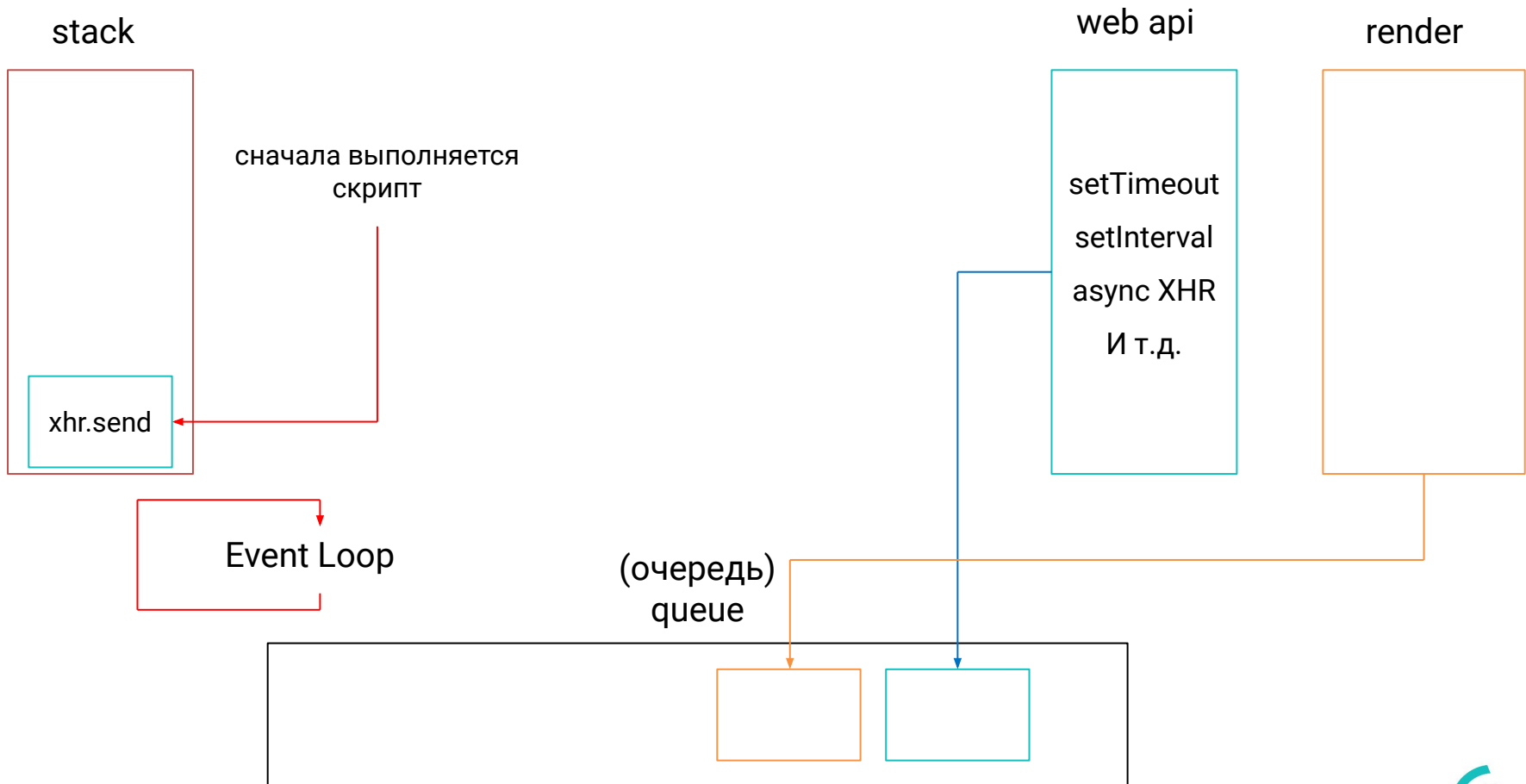
Это не ошибка. Это warning (предупреждение). В нём говорится о том, что не стоит в основном потоке делать синхронные запросы, потому что это ведёт к негативному пользовательскому опыту. В нашем случае – freeze.

А вот теперь нам детально нужно понять, что же происходит, и как этого избежать.



Event Loop

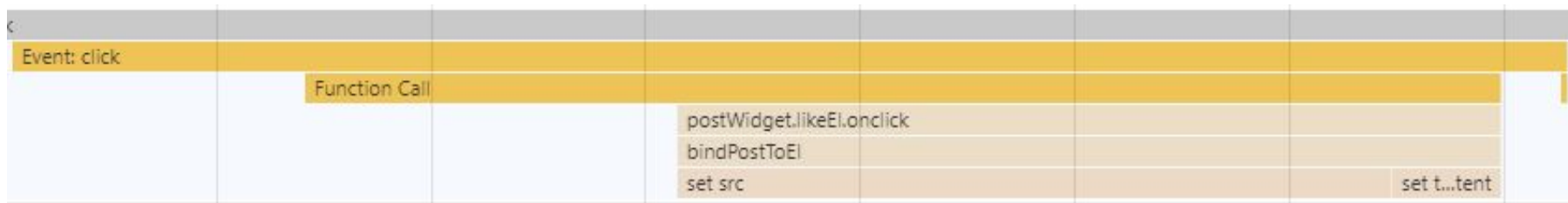
Вам нужно запомнить вот эту картинку:



Event Loop

В браузере есть цикл обработки событий. Одновременно браузер (мы с вами это уже обсуждали), может выполнять только одну функцию (речь о JS функции в конкретной вкладке).

Первое, что делает браузер – это запускает наш скрипт и выполняет его. Для этого он кладёт в `stack` (стэк) все вызовы функций, которые есть (мы с вами это видели, когда вызывали одну функцию из другой):



`onclick` вызвал `bindPostToEl`, а тот вызвал сначала `set src`, а затем `set textContent`.



Event Loop

Пока в стэке есть хоть одна функция, браузер ничего больше не делает – он занят выполнением этой функции.

В нашем случае, при загрузке скрипта, мы синхронно запускаем `xhr.send`, а это значит, что браузер занят выполнением этой функции и ничего больше не делает (а именно, не может ничего отрисовать).



Event Loop

Схитрим и перепишем код вот так:

```
setTimeout(() => {  
    const xhr = new XMLHttpRequest();  
    xhr.open('GET', apiUrl, false);  
    xhr.send();  
    console.log(xhr.response);  
}, 0);
```

Обратите внимание, мы поставили 0 – вам это может показаться бессмысленным, но это не так (откройте в новой вкладке). Теперь страница отрисовывается мгновенно, но потом замораживается (лучше запускать всё в инкогнито режиме - **Ctrl + Shift + N**, поскольку в нём по умолчанию отключены все плагины, которые могут повлиять на загрузку страницы).



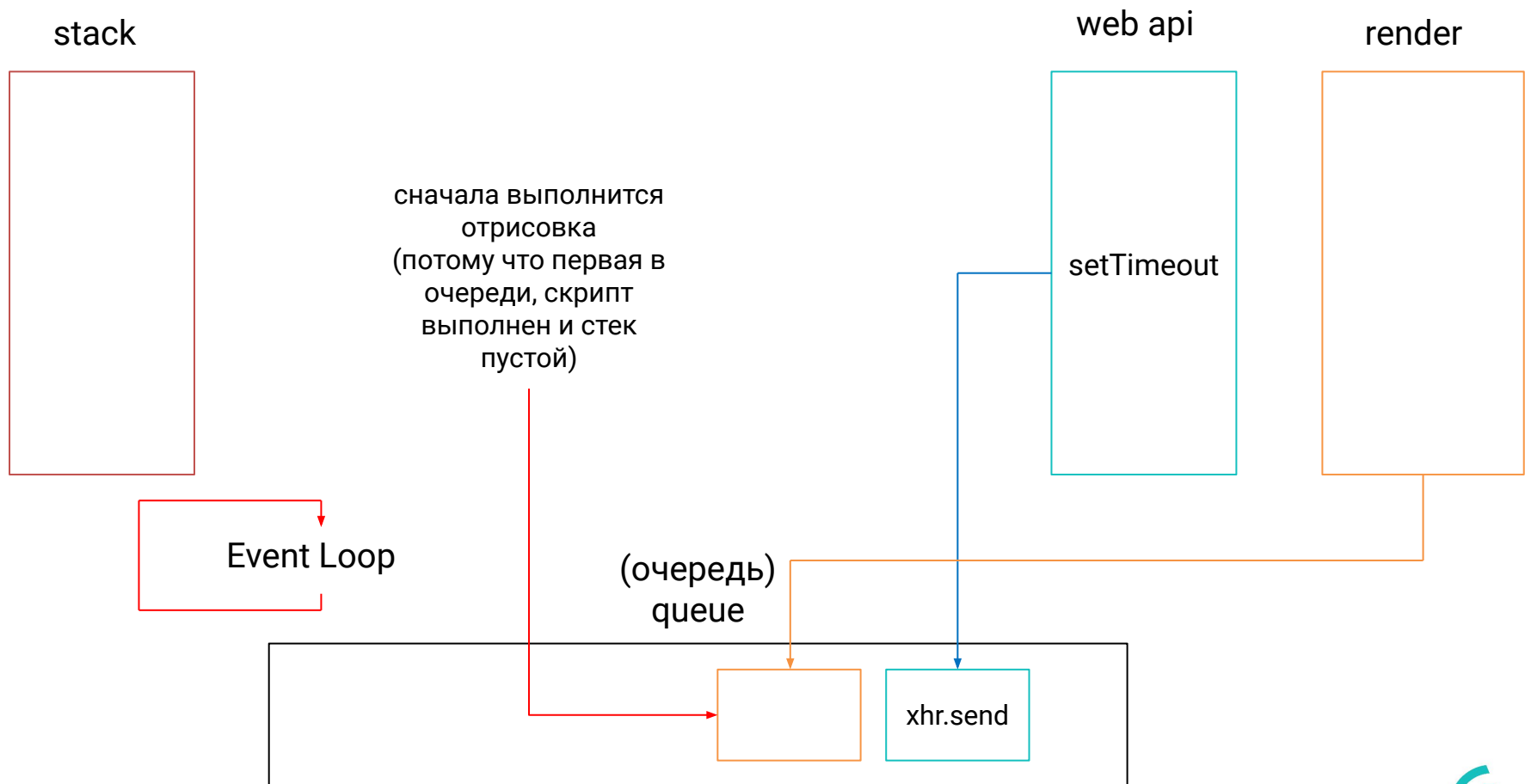
Event Loop

В чём же дело? Теперь скрипт обрабатывает мгновенно, а всё, что записано в `setTimeout` ставится в очередь (queue). При этом Renderer тоже ставит задачи по отрисовке в queue.

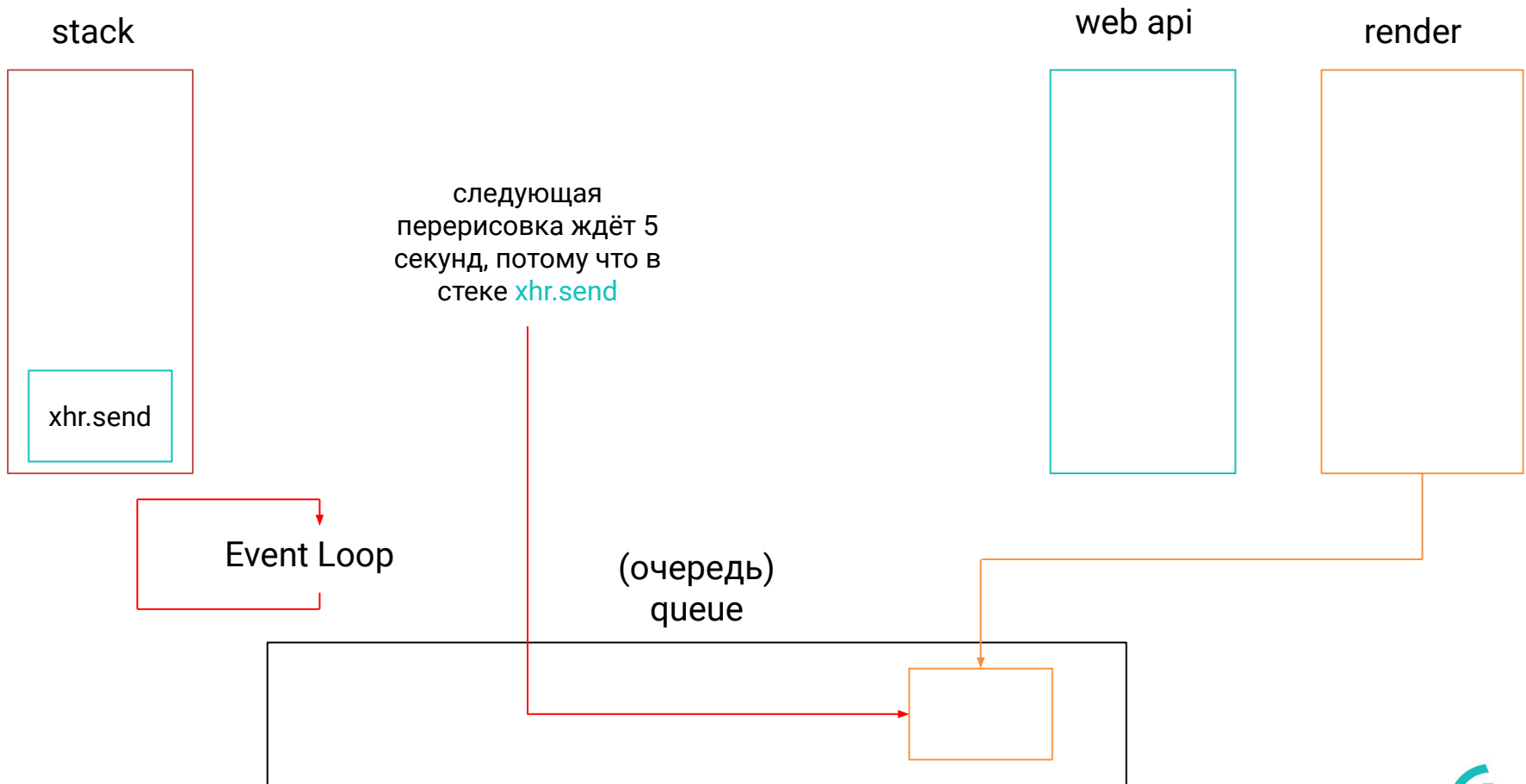
Браузер "вытаскивает" задачи из queue и выполняет. В нашем случае, он успел выполнить задачу Renderer, чтобы отрисовать интерфейс. Так вот, если в queue стоит выполнение кода (у нас код внутри `setTimeout`), то браузер этот код снова закидывает в стек и ждёт его выполнения.



Event Loop



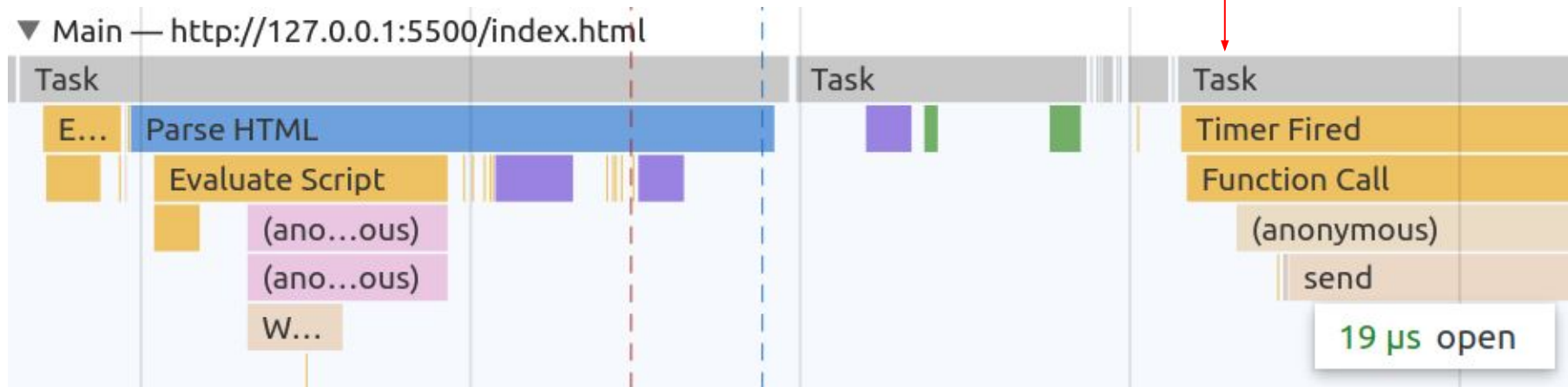
Event Loop



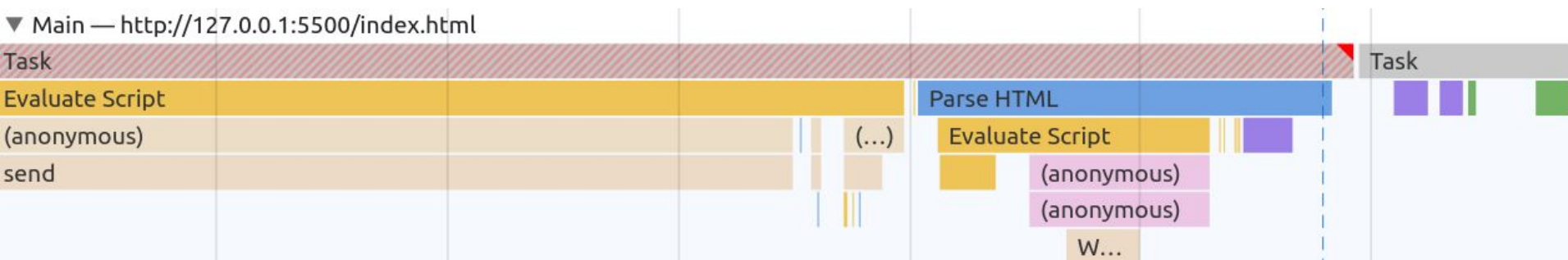
Event Loop

C `setTimeout`:

только тут сработал таймер:



Без `setTimeout` отрисовка (см. зелёные блоки) будет только после `send`:



Profiling

Напоминаем: чтобы получить такие "диаграммы" вам нужно перейти на вкладку Performance и дождаться окончания загрузки после клика на выделенную кнопку:



Event Loop

Про Event Loop обязательно нужно знать, т.к. без понимания, пусть даже в упрощённом виде, как у нас, трудно называться фронтенд-разработчиком.



Event Loop

Это всё здорово, но как же быть нам с нашим "медленным" сервером. Мы же не можем подойти к разработчику и попросить "всё переделать, чтобы работало быстро".

Для этого у нас есть асинхронная форма, работает она примерно так же, как и с обработчиками событий (кстати, события тоже ставятся в ту же самую очередь и ждут).



async XHR

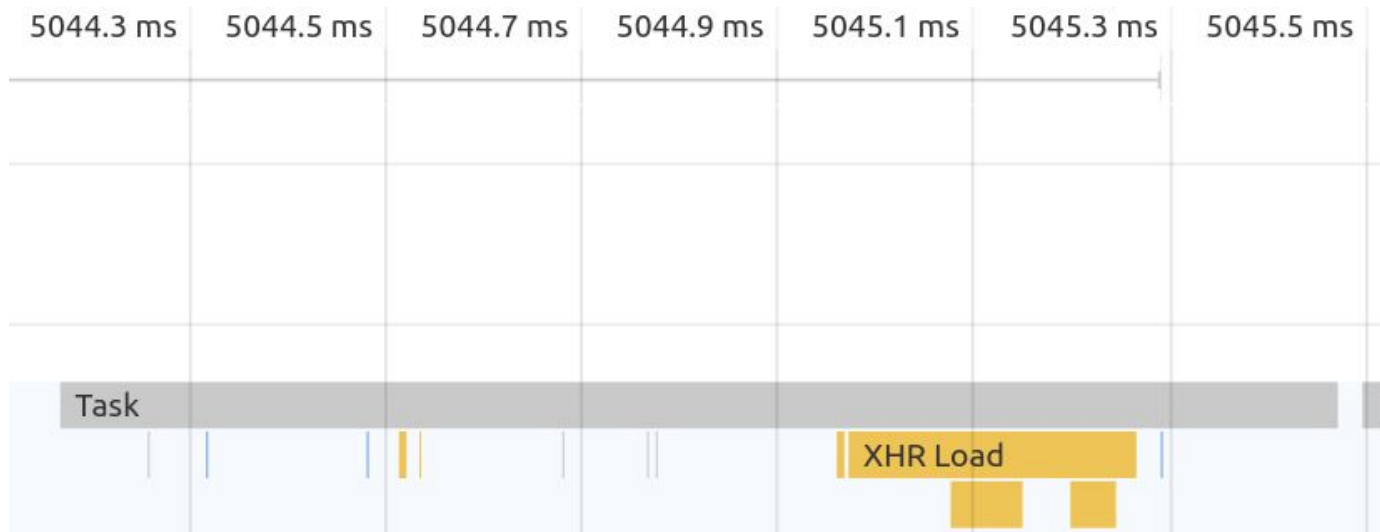
Для этого у нас есть асинхронная форма, работает она примерно так же, как и с обработчиками событий (кстати, события тоже ставятся в ту же самую очередь и ждут):

```
28  const apiUrl = 'http://127.0.0.1:9999/api/wishes';
29
30  const xhr = new XMLHttpRequest();
31  xhr.open('GET', apiUrl);
32  xhr.onload = evt => {
33    | // пришёл ответ
34  };
35  xhr.onerror = evt => {
36    | // ответ не пришёл
37  };
38  xhr.onloadend = evt => {
39    | // запрос завершился, с ошибкой или без
40  };
41  xhr.send();
```

В принципе, в комментариях всё указано, давайте дорабатывать. Начнём мы с конца: `onloadend` вызывается тогда, когда запрос завершился (не важно, успешно или нет). Это идеальное место, чтобы скрывать какой-нибудь индикатор загрузки.



async XHR



Это будет в самом конце (на 5-ой секунде).



onloadend

```
27  const loaderEl = rootEl.querySelector('[data-id="loader"]');
28
29  const apiUrl = 'http://127.0.0.1:9999/api/wishes';
30
31  const xhr = new XMLHttpRequest();
32  xhr.open('GET', apiUrl);
33  xhr.onload = evt => {
34    // пришёл ответ
35  };
36  xhr.onerror = evt => {
37    // ответ не пришёл
38  };
39  xhr.onloadend = evt => {
40    loaderEl.style.display = 'none';
41  };
42  xhr.send();
```



Load Data

А вообще, неплохо бы сделать отдельную функцию, которая отвечала бы за загрузку данных:

```
27  const loaderEl = rootEl.querySelector('[data-id="loader"]');
28
29  const apiUrl = 'http://127.0.0.1:9999/api/wishes';
30
31  function loadData(loaderEl) {
32      loaderEl.style.display = 'block';
33
34      const xhr = new XMLHttpRequest();
35      xhr.open('GET', apiUrl);
36      xhr.onload = evt => {
37          // пришёл ответ
38      };
39      xhr.onerror = evt => {
40          // ответ не пришёл
41      };
42      xhr.onloadend = evt => {
43          loaderEl.style.display = 'none';
44      };
45      xhr.send();
46  }
47
48  loadData(loaderEl);
```



onreadystatechange

При работе с XHR вы можете столкнуться с тем, что в некоторых статьях используется событие `readystatechange` и обработчик `onreadystatechange`. Это устаревшая форма, те события и обработчики, которые рассматриваем мы, на данный момент являются более предпочтительными.



onerror

`onerror` возникает тогда, когда происходит ошибка на уровне сети. Например, у вас пропал Интернет или сервер "просто умер". В этом случае, нам нужен элемент, в который бы мы выводили ошибку (как это делать, вы уже знаете, поэтому мы просто будем выводить ошибку в консоль):

```
39 | xhr.onerror = evt => {  
40 |     console.log(evt);  
41 | };
```



onerror

Теперь нужно сэмулировать ошибку. Для этого перейдите в терминал, где у вас работает сервер и нажмите **Ctrl + C**, чтобы он остановился и обновите страницу

app.js:40

```
▼ ProgressEvent ⓘ  
  bubbles: false  
  cancelBubble: false  
  cancelable: false  
  composed: false  
  ▶ currentTarget: XMLHttpRequest {onreadystatechange: null, readyState: 4, ...}  
  defaultPrevented: false  
  eventPhase: 0  
  isTrusted: true  
  lengthComputable: false  
  loaded: 0  
  ▶ path: []  
  returnValue: true  
  ▶ srcElement: XMLHttpRequest {onreadystatechange: null, readyState: 4, tim...  
  ▶ target: XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout...  
    timeStamp: 2050.5450000055134  
    total: 0  
    type: "error"  
  ▶ __proto__: ProgressEvent
```

✖ Failed to load resource: net::ERR_CONNECTION_REFUSED :9999/api/wishes:1

Красную часть печатаем не мы, её печатает сам браузер.



Network

Вы можете посмотреть, что происходит, на вкладке Network (поставьте переключатель в положение XHR, чтобы показывались только XHR запросы):

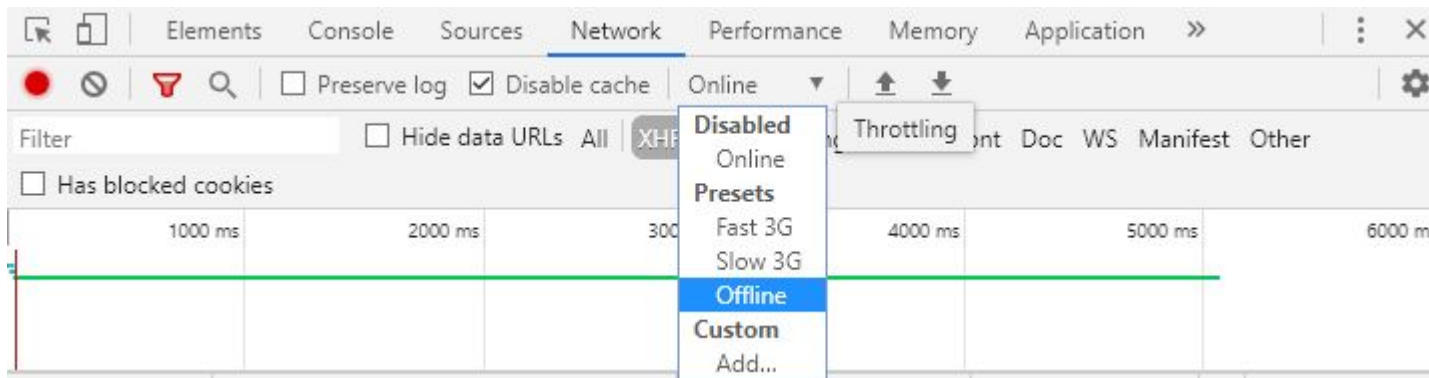
The screenshot shows the Chrome DevTools Network tab. The top bar includes tabs for Elements, Console, Sources, Network (selected), Performance, Memory, and Application. Below the tabs, there are icons for a red circle, a crossed-out circle, a funnel, and a magnifying glass. A row of checkboxes includes 'Preserve log' (unchecked), 'Disable cache' (checked), and 'Online' (selected). A filter bar shows 'Filter' with a text input, 'Hide data URLs' (unchecked), and a list of request types: 'All', 'XHR' (selected), 'JS', 'CSS', 'Img', 'Media', 'Font', 'Doc', 'WS', and 'Manifest'. Below the filter bar, there is a checkbox for 'Has blocked cookies'. A timeline at the top shows a duration from 0 to 2000 ms with markers at 500 ms, 1000 ms, 1500 ms, and 2000 ms. A blue bar represents the network activity, starting at 0 ms and ending at approximately 1800 ms. Below the timeline, a table lists the network requests.

Name	Status	Type	Initiator	Size	T...
<input type="checkbox"/> wishes	(failed) net::ERR_CONNECTION_REFUSED	xhr	app.js:45	0 B	2...



onerror

Запустите снова сервер (`node server.js` в консоли). Мы можем сэмулировать и отсутствие сети и у нас (не обязательно отключать сервер):



Но на данный момент это не принесёт желаемого результата, т.к. при обновлении просто не загрузится вся страница. К этому вопросу мы вернёмся чуть позже.



onload

`onload` срабатывает тогда, когда нам от сервера приходит ответ, причём не важно какой. Что значит не важно какой? На самом деле, сервер нам может ответить "ошибкой". Например, он может нам сказать, что мы делаем запрос не туда или не так.

Для того, чтобы это проверить, поменяйте `apiUrl` на следующий:

```
29  const apiUrl = 'http://127.0.0.1:9999/api/wishlist';
```

А код обработчика на:

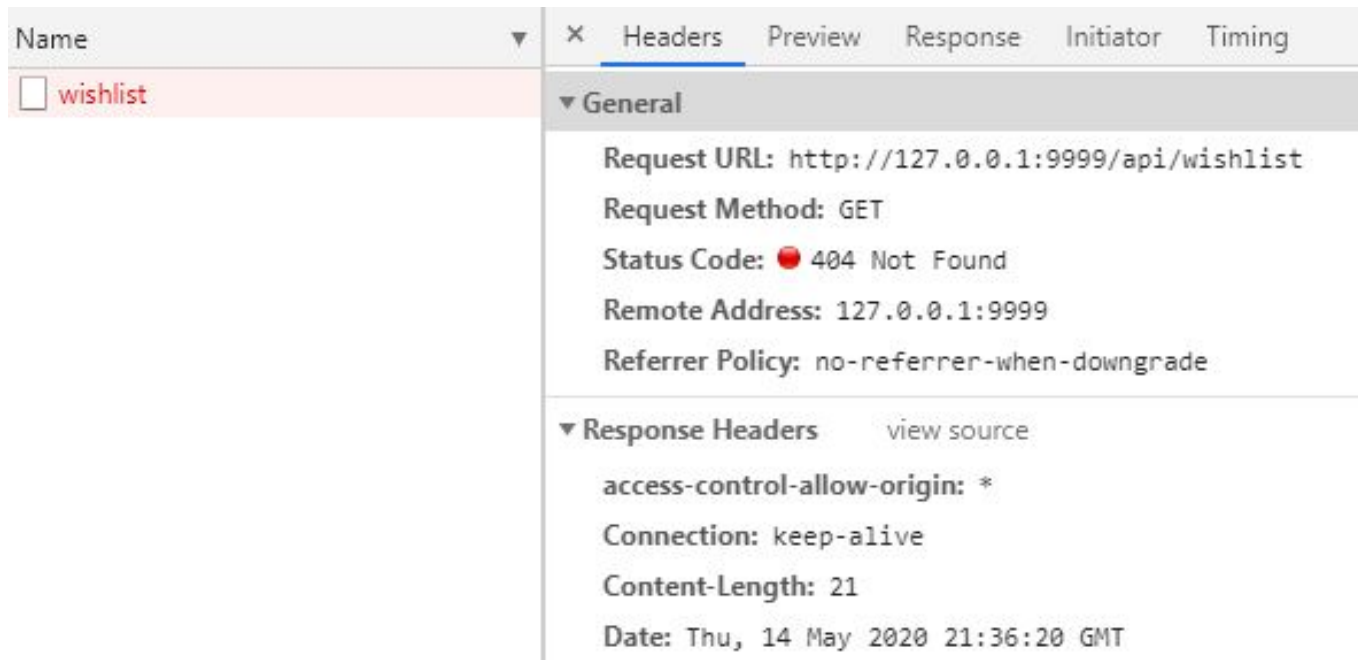
```
36  |   xhr.onload = evt => {  
37  |       console.log(evt);  
38  |   };
```

После чего обновите страницу.



status codes

Наверняка вы слышали шутки про 404:



Но давайте разберёмся, что это такое. Одной из частей response message в HTTP является Status Line. Вот туда прописывается числовой код ответа, содержащий информацию о том, насколько успешно выполнен запрос.



status codes

Статус коды объединены в следующие группы:

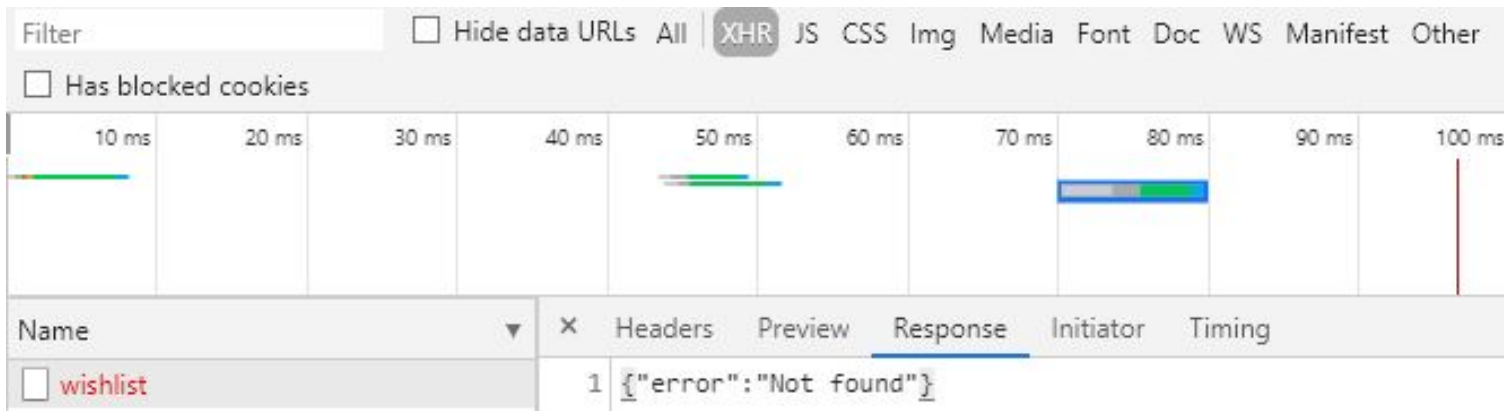
- 100-199 – Info (информационные)
- 200-299 – Success (успешно)
- 300-399 – Redirection (перенаправление)
- 400-499 – Client Error (ошибка клиента)
- 500-599 – Server Error (ошибка сервера)

Обратите внимание, несмотря на то, что 404 – это ошибка клиента, в XMLHttpRequest срабатывает **onload**, поскольку данные пришли.



response

Давайте посмотрим на ответ, на вкладке Response:



Очень похоже на JS объект, не так ли?



JSON



JSON

JSON (JavaScript Object Notation) – это специальный текстовый формат передачи данных, один из самых популярных на сегодняшний день.

В чём суть: программные системы могут быть написаны на разных языках, например, мы можем писать фронтенд на JS, а мобильные приложения пишутся на Kotlin или Swift. Сервер вообще может быть написан на Java. При этом всем этим системам нужно обмениваться данными. JSON – как раз-таки такой формат, который позволяет это делать.

Нам не с вами пока не нужно знать про внутреннее устройство JSON (хотя мы настоятельно рекомендуем с ним ознакомиться) по одной простой причине: у нас с вами есть глобальный объект **JSON** с двумя методами – **parse** (из JSON документа создаёт JS объект) и **stringify** (из JS объекта создаёт JSON документ).



JSON

В JSON могут присутствовать следующие типы данных:

1. Объекты (имена свойств должны быть в двойных кавычках)
2. Массивы
3. Строки (должны быть в двойных кавычках)
4. Числа
5. Boolean (true и false)
6. Null (null)

Других типов не разрешено. Т.е. `undefined` нельзя, кроме того, нельзя функции, методы и т.д. JSON служит только для передачи данных. Этого нам должно быть достаточно для работы. Смотрим.



JSON

```
36  xhr.onload = evt => {  
37      if (xhr.status < 200 || xhr.status > 299) {  
38          // произошла ошибка  
39          const error = JSON.parse(xhr.responseText);  
40          console.log(error);  
41          return;  
42      }  
43  
44      const data = JSON.parse(xhr.responseText);  
45      console.log(data);  
46  };
```

Строка 37 – смотрим на статус код (он кладётся в объект `xhr`) – если не 2xx, значит сервер прислал ошибку.

`||` - логический оператор или, возвращает `true` тогда, когда хотя бы один из операндов `true`*. В пару к нему есть логический оператор `&&` (и), который возвращает `true` только тогда, когда оба операнда `true`*

Примечание*: с не `boolean` значениями это работает похитрее, но вы узнаете об этом из курса React.



JSON

```
36  xhr.onload = evt => {  
37      if (xhr.status < 200 || xhr.status > 299) {  
38          // произошла ошибка  
39          const error = JSON.parse(xhr.responseText);  
40          console.log(error);  
41          return;  
42      }  
43  
44      const data = JSON.parse(xhr.responseText);  
45      console.log(data);  
46  };
```

Строка 39 – "парсим" ответ. Тело ответа в виде строки (то, что мы видели на вкладке Response), кладётся в свойство `responseText` объекта `xhr`.

Строка 40-41 – логируем ошибку и выходим (вы можете отобразить в DOM элементе)

Строка 44 – если статус код 2xx, то парсим ответ и складываем в переменную `data`.

Важно: в данном случае мы не усложняем код и считаем, что сервер выдаёт "правильные" JSON ответы. О том, что делать с неправильными – на следующих курсах.



Код 200

В большинстве статей в интернете вы можете встретить вот такую проверку:





```
if (xhr.status !== 200) {  
    // ошибка  
}
```

Этот подход не совсем верен, потому что по спецификации успешными считаются коды 2xx. Но опять же тут всё зависит от того, как написан сервер и какие ответы он выдаёт.



Данные

Поправим `apiUrl` обратно на <http://127.0.0.1:9999/api/wishes> и посмотрим, что нам приходит на вкладке Response:

Name	×	Headers	Preview	Response	Initiator	Timing
 index.html	1					
 styles.css						
 app.js						
 wishes						

Пустой массив. Поскольку желаний у нас ещё на сервере нет, то и массив пустой.

Как раз это значение мы и будем использовать для инициализации.



Загрузка данных

Давайте создадим объект, в полях которого будем хранить все данные нашего приложения (1).

Благодаря такому подходу, мы можем передавать этот объект (2) и менять его поля (3).

```
31  const state = {
32    |   wishes: [],
33  };
34
35  function loadData(loaderEl, state) {
36    loaderEl.style.display = 'block';
37
38    const xhr = new XMLHttpRequest();
39    xhr.open('GET', apiUrl);
40    xhr.onload = evt => {
41      if (xhr.status < 200 || xhr.status > 299) {
42        // произошла ошибка
43        const error = JSON.parse(xhr.responseText);
44        console.log(error);
45        return;
46      }
47
48      state.wishes = JSON.parse(xhr.responseText);
49    };
50  >  xhr.onerror = evt => { ...
51    };
52  >  xhr.onloadend = evt => { ...
53    };
54    xhr.send();
55  }
56
57
58
59  loadData(loaderEl, state);
```

Отрисовка данных

Несмотря на то, что данных у нас пока нет, мы уже можем вставить код, который будет отвечать за их отрисовку. А это значит, что нам нужно внутри обработчика `onload` вставить ещё код. И потихоньку мы приходим к тому, что функция `loadData` становится очень сложной. А самое нехорошее – она слишком много знает. Она знает и про `loader`, и про состояние, а сейчас будет знать ещё и про отрисовку данных. Хотя её задача – просто загружать данные.

Как бы так сделать, чтобы она знала "поменьше"? Давайте посмотрим на аналогии: как `setTimeout` или `setInterval` узнают, какую функцию выполнять? Мы просто передаём эту функцию в качестве параметра. Почему бы нам не поступить так же с `loadData`?



callbacks

```
35 function loadData(callbacks) {  
36     if (typeof callbacks.onStart === 'function') {  
37         callbacks.onStart();  
38     }  
39  
40     const xhr = new XMLHttpRequest();  
41     xhr.open('GET', apiUrl);  
42     xhr.onload = () => {  
43         if (xhr.status < 200 || xhr.status > 299) {  
44             const error = JSON.parse(xhr.responseText);  
45             if (typeof callbacks.onError === 'function') {  
46                 callbacks.onError(error);  
47             }  
48             return;  
49         }  
50  
51         const data = JSON.parse(xhr.responseText);  
52         if (typeof callbacks.onSuccess === 'function') {  
53             callbacks.onSuccess(data);  
54         }  
55     }  
};
```



callbacks

```
56 | xhr.onerror = () => {  
57 | |   if (typeof callbacks.onError === 'function') {  
58 | |     callbacks.onError({error: 'network error'});  
59 | |   }  
60 | | };  
61 | xhr.onloadend = () => {  
62 | |   if (typeof callbacks.onFinish === 'function') {  
63 | |     callbacks.onFinish();  
64 | |   }  
65 | | };  
66 | xhr.send();  
67 | }
```



callbacks

Давайте разбираться: мы хотим передавать в функцию `loadData` объект `callbacks`, в котором может быть (но не обязательно должно) 4 функции:

- `onStart` – чтобы запускать при старте запроса
- `onFinish` – чтобы запускать при завершении запроса
- `onSuccess` – чтобы запускать при успешном получении данных (код 2xx)
- `onError` – чтобы запускать при ошибке (код не 2xx или сетевая ошибка)

Вот так мы это будем вызывать:

```
69  loadData({
70    onStart: () => loaderEl.style.display = 'block',
71    onFinish: () => loaderEl.style.display = 'none',
72    onSuccess: data => {
73      state.wishes = data;
74      // TODO: отрисовать данные
75    },
76    onError: error => console.log(error),
77  });
```

Немного сложновато,
но привыкнуть можно



callbacks

Таким образом, теперь функция `loadData` ничего не знает, кроме того, что ей нужно запускать callback'и, если они ей переданы (а если не переданы, то условия в `if`'ах не сработают и ничего страшного не случится).



Отрисовка элементов

Осталось отрисовать наши элементы. Напишем для этого функцию `renderWishes`:

```
70 function renderWishes(wishesEl, wishes) {  
71     wishesEl.innerHTML = wishes.map(o => `  
72         <li>  
73             <div>Название: ${o.name}, цена: ${o.price} с. <button data-action="remove">Удалить</button></div>  
74             <div data-block="description">${o.description}</div>  
75         </li>  
76     `).join('');  
77 }
```

Давайте разбираться: `wishes` – это массив (пусть даже пустой), значит у него есть метод `map`, который из массива объектов делаем массив строк. У массива, кроме того, есть метод `join`, который позволяет все элементы склеить в строку. В итоге мы получаем одну большую строку и кладём её в `innerHTML`, а браузер сам всё парсит и создаёт элементы.

При этом у `innerHTML` есть важная особенность: если у `wishesEl` до этого были дети, то теперь они заменятся на новых (а старые браузер удалит).



Отрисовка элементов

Выбираем `wishesEl` (рядом с `loader`'ом):

```
27  const loaderEl = rootEl.querySelector('[data-id="loader"]');
28  const wishesEl = rootEl.querySelector('[data-id="wish-list"]');
```

И вызываем нашу функцию для отрисовки:

```
79  loadData({
80    onStart: () => loaderEl.style.display = 'block',
81    onFinish: () => loaderEl.style.display = 'none',
82    onSuccess: data => {
83      state.wishes = data;
84      renderWishes(wishesEl, state.wishes);
85    },
86    onError: error => console.log(error),
87  });
```



Отрисовка элементов

Конечно же, немного глупо писать код, работу которого нельзя проверить. Поэтому мы для вас подготовили специальный демо-набор желаний. Доступен он по адресу

<http://127.0.0.1:9999/demo/api/wishes>:

- Название: MacBook Pro, цена: 37000 с.
Мощное железо
- Название: Chevrolet Camaro, цена: 500000 с.
Стильное авто

Конечно же, нужно будет ещё вызвать функцию пересчёта общей стоимости, но надеемся, что с этим вы справитесь сами.



Добавление



Отрисовка элементов

Разберёмся с добавлением. Программист серверной части утверждает, что для добавления нужно:

1. Отправить POST-запрос на адрес <http://127.0.0.1:9999/api/wishes>
2. Установить заголовок `Content-Type` в значение `application/json`
3. Передать в теле запроса JSON следующего вида:

```
{  
  "id": 0,  
  "name": "Ваше название",  
  "price": 1000,  
  "description": "Ваше описание"  
}
```

В ответ вернётся JSON с проставленным id.



Добавление

Разберёмся
сначала с формой:

Мы не стали
дублировать
проверки из
прошлой лекции, вы
можете их добавить
самостоятельно.

```
93  const formEl = rootEl.querySelector('[data-id="wish-form"]');
94  const nameEl = formEl.querySelector('[data-input="name"]');
95  const priceEl = formEl.querySelector('[data-input="price"]');
96  const descriptionEl = formEl.querySelector('[data-input="description"]');
97  formEl.onSubmit = evt => {
98      evt.preventDefault();
99
100     const id = 0;
101     const name = nameEl.value.trim();
102     const price = Number(priceEl.value);
103     const description = descriptionEl.value.trim();
104     // TODO: проверки из прошлой лекции
105     const wish = {
106         id,
107         name,
108         price,
109         description,
110     };
```



Функция saveData

```
89 function saveData(item, callbacks) {  
90     if (typeof callbacks.onStart === 'function') {  
91         callbacks.onStart();  
92     }  
93  
94     const xhr = new XMLHttpRequest();  
95     xhr.open('POST', apiUrl);  
96 >     xhr.onload = () => { ...  
109     };  
110 >     xhr.onerror = () => { ...  
114     };  
115 >     xhr.onloadend = () => { ...  
119     };  
120     xhr.setRequestHeader('Content-Type', 'application/json');  
121     xhr.send(JSON.stringify(item));  
122 }
```

Свёрнутые обработчики аналогичны тем, что были в функции `loadData`. Красным отмечены изменения. Возникает вопрос "что это за заголовки такие и для чего они нужны"? Заголовки – это мета-данные, по которым сервер и клиент "договариваются" о том, что они друг другу передают. Заголовок `Content-Type`, отправляемый клиентом, сообщает о том, что мы передаём JSON документ.



JSON и не JSON

Передавать можно не только JSON. В рамках бонусной лекции вы познакомитесь ещё с несколькими возможностями передачи данных, в том числе научитесь загружать медиа-файлы (фото, аудио и видео) на сервер.



Вызываем saveData

В целом, всё очень похоже на загрузку данных.

Ключевые моменты:

1. `renderWishes` заново перерисует все желания (если их будет много – скролл на странице будет "прыгать")
2. Сброс формы стоит в `onSuccess`
3. Мы не отключаем элементы формы на время загрузки

```
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152

const id = 0;
const name = nameEl.value.trim();
const price = Number(priceEl.value);
const description = descriptionEl.value.trim();
// TODO: проверки из прошлой лекции
const wish = {
  id,
  name,
  price,
  description,
};

saveData(wish, {
  onStart: () => loaderEl.style.display = 'block',
  onFinish: () => loaderEl.style.display = 'none',
  onSuccess: data => {
    state.wishes.unshift(data);
    renderWishes(wishesEl, state.wishes);
    formEl.reset();
  },
  onError: error => console.log(error),
});
```



Детали

Иногда возникает необходимость "заблокировать" форму на то время, пока данные из неё отправляются. Не всегда это нужно, потому что в продвинутых интерфейсах считается лучшим не блокировать, а дать пользователю вводить новые желания дальше.

Мы пойдём по-простому пути и попробуем заблокировать. К сожалению, свойство `disabled` есть только у полей ввода и кнопкой. Целиком форму заблокировать нельзя.

Но есть обходной путь – элемент `fieldset`. Он позволяет объединить в себе набор полей для ввода и кнопки. И у него есть свойство `disabled`. Если его выставить в `true`, то отключаться все содержащиеся в нём поля формы.



Детали

```
4 rootEl.innerHTML = `
5   <h1>WishList</h1>
6   <div data-id="loader">Загружаем данные...</div>
7   <form data-id="wish-form">
8     <fieldset data-id="wish-fieldset">
9       <div data-id="message"></div>
10      <div>
11        <label for="name-input">Название</label>
12        <input data-input="name" id="name-input">
13      </div>
14      <div>
15        <label for="price-input">Цена</label>
16        <input data-input="price" id="price-input" type="number">
17      </div>
18      <div>
19        <label for="description-input">Описание</label>
20        <textarea data-input="description" id="description-input" rows="5"></textarea>
21      </div>
22      <button>Добавить</button>
23    </fieldset>
24  </form>
25  <div>Необходимо: <span data-id="total">0</span> с.</div>
26  <ul data-id="wish-list"></ul>
27 `;
```



Детали

```
126 const formEl = rootEl.querySelector('[data-id="wish-form"]');
127 → const fieldsetEl = formEl.querySelector('[data-id="wish-fieldset"]');
128 const nameEl = formEl.querySelector('[data-input="name"]');
129 const priceEl = formEl.querySelector('[data-input="price"]');
130 const descriptionEl = formEl.querySelector('[data-input="description"]');
```

```
146   saveData(wish, {
147     onStart: () => {
148       loaderEl.style.display = 'block';
149 →   fieldsetEl.disabled = true;
150     },
151     onFinish: () => {
152       loaderEl.style.display = 'none';
153 →   fieldsetEl.disabled = false;
154     },
155     onSuccess: data => {
156       state.wishes.unshift(data);
157       renderWishes(wishesEl, state.wishes);
158       formEl.reset();
159     },
160     onError: error => console.log(error),
161   });
```

Важно: не забудьте из [apiUrl](#) убрать demo



Удаление



Удаление

Разберёмся с удалением. Программист серверной части утверждает, что для удаления нужно: отправить **DELETE**-запрос на адрес <http://127.0.0.1:9999/api/wishes/{id}>, где вместо **{id}** передать идентификатор элемента, который мы собираемся удалить (это часто употребляемая нотация, иногда вы также будете встречать <http://127.0.0.1:9999/api/wishes/:id> с таким же смыслом для **:id**).

Это достаточно частый приём – передавать идентификатор в пути запроса. И вроде бы всё просто, ни тела передавать не нужно, ни заголовков. Но есть одна загвоздка: как навесить обработчики на кнопки удалить?



Удаление

Первый вариант достаточно тривиальный и выглядит он так:

```
function renderWishes(wishesEl, wishes) {  
  wishes.map(o => {  
    const el = document.createElement('li');  
    el.innerHTML = `  
      <div>Название: ${o.name}, цена: ${o.price} с. <button data-action="remove">Удалить</button></div>  
      <div data-block="description">${o.description}</div>  
    `;  
    el.querySelector('[data-action="remove"]').onclick = () => {  
      // TODO: функция удаления  
    };  
    return el;  
  }).forEach(o => wishesEl.appendChild(o));  
}
```

Но как-то это не очень красиво и интересно. Давайте немного поговорим о событиях.

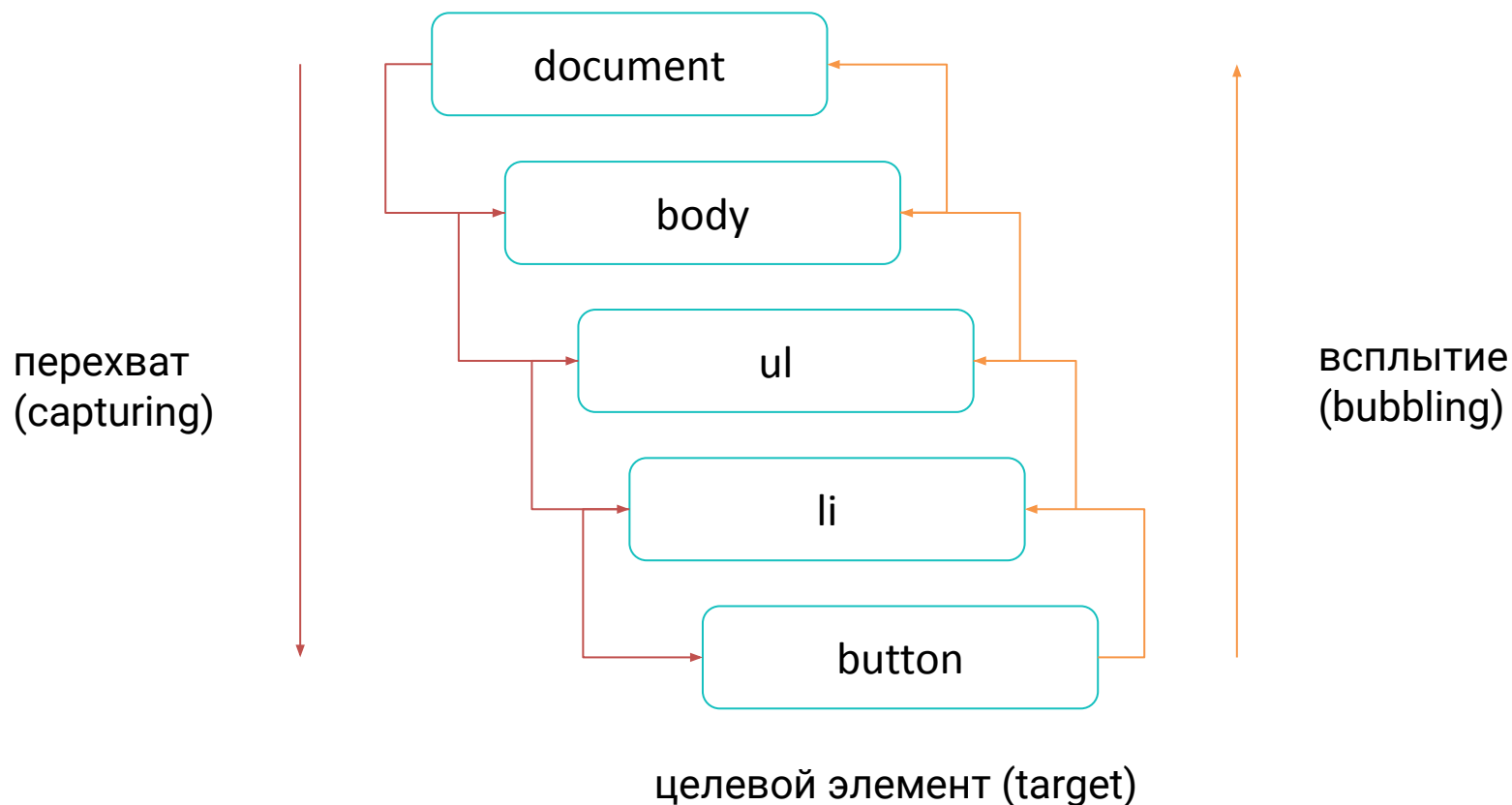


События



События

Обработка событий в браузере устроена немного сложнее, чем мы с вами говорили. В целом, она проходит в три этапа и выглядит вот так:



События

Т.е. когда мы кликаем на кнопку удалить, событие клика сначала сверху вниз проходит через документ и всех родителей, затем срабатывает на самом элементе, а потом поднимается вверх до документа.

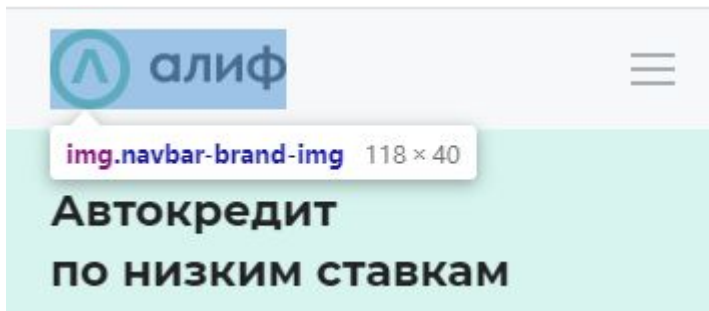
Мы не будем сейчас вдаваться в историю и выяснять почему так, просто примем за данность.



События

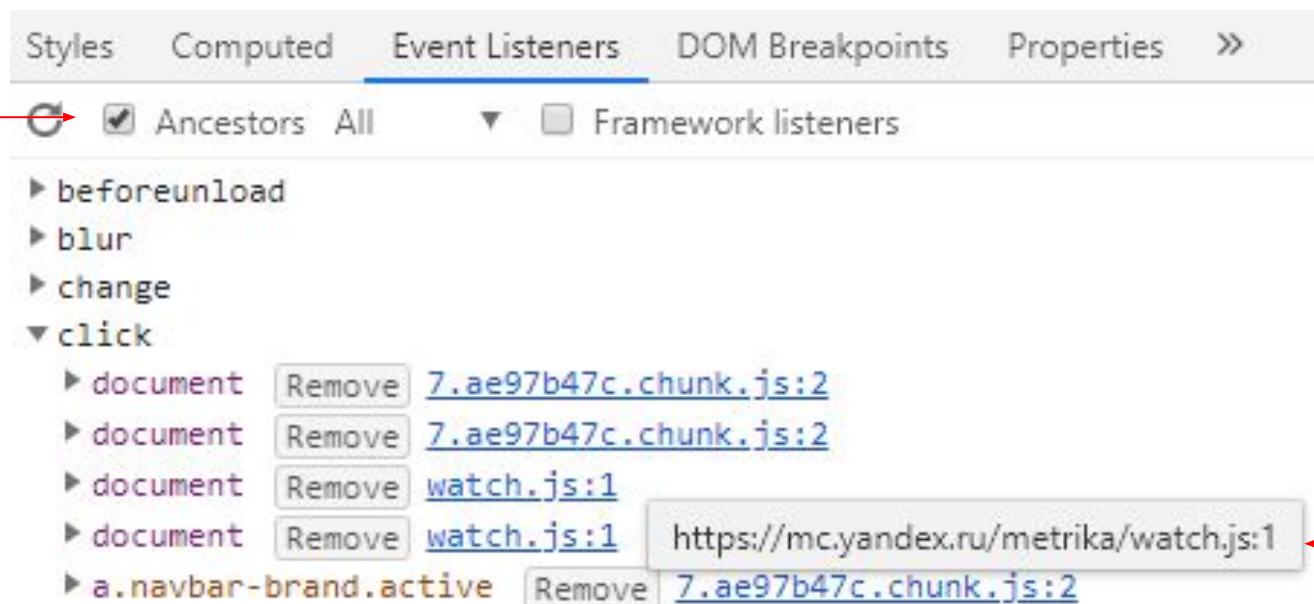
Первый вопрос – как это использовать? Это широко используется системами аналитики, например, Яндекс Метрикой или Google Analytics.

Зайдём на сайт <https://alif.tj> и выберем, например, логотип:



События

В панельке Elements перейдём на вкладку Event Listeners и поставим флажок Ancestors:



События

Т.е. обработчик клика установлен на документе и когда вы кликаете на логотип, Яндекс.Метрика это видит. И что самое интересное, работает он на фазе Capture, чтобы перехватить этот клик раньше всех:

```
▼ click
  ► document Remove 7.ae97b47c.chunk.js:2
  ► document Remove 7.ae97b47c.chunk.js:2
  ► document Remove watch.js:1
  ▼ document Remove watch.js:1
    useCapture: true
    passive: true
    once: false
    ► handler: f ()
  ► a.navbar-brand.active Remove 7.ae97b47c.chunk.js:2
```



События

Если мы посмотрим на наши обработчики, то увидим, что они работают на совсем другой фазе:

```
▼ submit
  ▼ form Remove app.js:142
    useCapture: false
    passive: false
    once: false
    ▶ handler: evt => {...}
```



События

Почему так? Всё дело в том, что "навешивать" обработчики событий можно тремя способами:

1. Через атрибут `on...` (например, `onclick`) прямо в HTML (не делайте так – это считается дурным тоном), хотя разработчики Vk так делают:

```
▼ <div class="post_date">  
  <a class="wall_text_name_explain_promoted_post  
    post_link" href="/wall-167497178_115102"  
    onclick="return showWiki({w: 'wall-  
    167497178_115102'}, false, event, {ads_params:
```

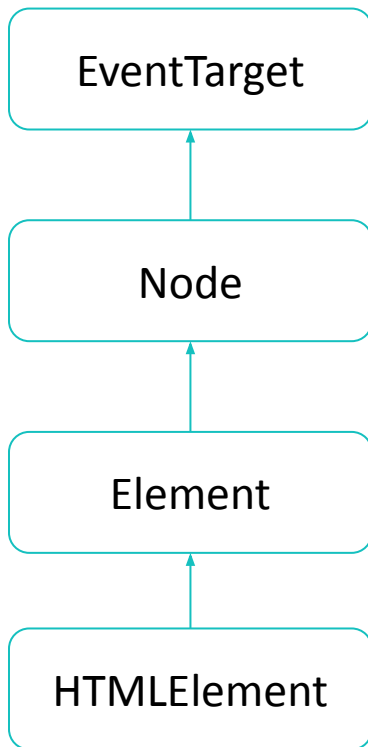
2. Через свойства `on...` (как мы до этого делали)
3. Через метод `addEventListener`



События

С первыми двумя всё понятно, давайте разбираться с третьим, откуда он взялся?

Если вы помните, то у нас была вот такая картинка:



Интерфейс `EventTarget` как раз и определяет три метода, позволяющие работать с событиями:

- `addEventListener()` – добавление
- `removeEventListener()` – удаление
- `dispatchEvent()` – генерация события из скрипта и отправка

`EventTarget`’ом могут быть не только DOM ноды, например, XHR – тоже `EventTarget`.



События

Так вот через этот интерфейс можно устанавливать обработчики на фазу Capture, кроме того, на одно и то же событие на одном и том же элементе можно навесить много обработчиков, в то время как через Event Handler'ы (то, что мы использовали) можно только один и только на фазу Bubbling.

Использование EventListener'ов считается более предпочтительным. Давайте же научимся ими пользоваться.

Провернём хитрый трюк – поскольку событие проходит через всех родителей, почему бы нам не повесить обработчик клика прямо на `ul`?



Bubbles

Нужно отметить, что не все события всплывают, если свойство `bubbles` – `false`, то событие не всплывает.

Interface	<code>InputEvent</code>
Sync / Async	Sync
Bubbles	Yes
Cancelable	No
Composed	Yes
Target	<code>Element</code>
Default Action	None



События

Для начала вернём нашу функцию `renderWishes` в прежнее состояние:

```
72 function renderWishes(wishesEl, wishes) {  
73   wishesEl.innerHTML = wishes.map(o => `  
74     <li>  
75       <div>Название: ${o.name}, цена: ${o.price} с. <button data-action="remove">Удалить</button></div>  
76       <div data-block="description">${o.description}</div>  
77     </li>  
78   `).join('');  
79 }
```

Установим обработчики и слушатели разными способами и посмотрим, в каком порядке они будут работать (и будут ли):

```
91 wishesEl.onclick = evt => {  
92   console.log(evt);  
93 };  
94  
95 wishesEl.addEventListener('click', evt => {  
96   console.log(evt);  
97 }); // для фазы Bubbling  
98  
99 wishesEl.addEventListener('click', evt => {  
100   console.log(evt);  
101 }, true); // для фазы Capturing
```



Порядок срабатывания

Кликаем на кнопку удалить и смотрим в консоль:

```
► MouseEvent {isTrusted: true, screenX: 513, screenY: 515, clientX: 315, clientY: 296, ...} app.js:100
► MouseEvent {isTrusted: true, screenX: 513, screenY: 515, clientX: 315, clientY: 296, ...} app.js:92
► MouseEvent {isTrusted: true, screenX: 513, screenY: 515, clientX: 315, clientY: 296, ...} app.js:96
```

```
91 wishesEl.onclick = evt => {
92   |   console.log(evt);
93   | };
94
95 wishesEl.addEventListener('click', evt => {
96   |   console.log(evt);
97   | }); // для фазы Bubbling
98
99 wishesEl.addEventListener('click', evt => { ←
100   |   console.log(evt);
101   | }, true); // для фазы Capturing
```

A red line originates from the closing curly brace of the event listener on line 99. It extends horizontally to the right, then turns vertically upwards, and finally turns horizontally to the left, ending with an arrowhead pointing to the opening curly brace of the event listener on line 99. This diagram illustrates that the event listener on line 99 (the Capturing phase) executes first, before the event listener on line 97 (the Bubbling phase).

Т.е. самым первым сработал слушатель с 99 строки, затем они начали работать в том порядке, в котором мы установили (можете поменять местами и удостоверится).



События

Оставим только вот этот слушатель (принято без особой необходимости не вешать слушатель на фазу Capturing):

```
91 wishesEl.addEventListener('click', evt => {  
92   |   console.log(evt);  
93   }); // для фазы Bubbling
```

Остаётся выяснить, как узнать, что кликнули именно в кнопку удаления, т.к. сейчас эта функция срабатывает вне зависимости от того, в каком месте списка мы кликнули.



target vs currentTarget

У объекта события (наш `evt`) есть два замечательных свойства:

1. `target` – это объект, на котором реально произошло событие (в кого мы физически кликнули – если в кнопку, то кнопка)
2. `currentTarget` – это объект, на котором сейчас выполняется обработчик (т.е. для кого мы сделали `onclick` или `addEventListener('click', ...)` – в нашем случае `wishesEl`)

Так вот именно через это мы и можем узнать, кликнули ли в кнопку, а если в кнопку, то узнать, в какую именно.



target vs currentTarget

Немного схитрим и добавим в кнопку атрибут, в котором будет храниться **id** желания:

```
function renderWishes(wishesEl, wishes) {  
  wishesEl.innerHTML = wishes.map(o => `  
    <li>  
      <div>Название: ${o.name}, цена: ${o.price} с. <button data-itemid="${o.id}" data-action="remove">Удалить</button></div>  
      <div data-block="description">${o.description}</div>  
    </li>  
  `).join('');  
}
```



target vs currentTarget

```
91 wishesEl.addEventListener('click', evt => {
92   if (evt.target.dataset.action !== 'remove') { ← Если не remove, значит не кнопка, выходим
93     return;
94   }
95
96   const id = Number(evt.target.dataset.itemid); ← Всё, что положили в data-атрибуты – строка,
97                                                    нам нужно число
98   removeDataById(id, {
99     onStart: () => loaderEl.style.display = 'block',
100    onFinish: () => loaderEl.style.display = 'none',
101    onSuccess: () => {
102      state.wishes = state.wishes.filter(o => o.id !== id);
103      renderWishes(wishesEl, state.wishes);
104    },
105    onError: error => console.log(error),
106  })
107 });
```



Функция removeDataById

```
91 function removeDataById(id, callbacks) {  
92     if (typeof callbacks.onStart === 'function') {  
93         callbacks.onStart();  
94     }  
95  
96     const xhr = new XMLHttpRequest();  
97     xhr.open('DELETE', `${apiUrl}/${id}`);  
98     xhr.onload = () => {  
99         if (xhr.status < 200 || xhr.status > 299) {  
100             const error = JSON.parse(xhr.responseText);  
101             if (typeof callbacks.onError === 'function') {  
102                 callbacks.onError(error);  
103             }  
104             return;  
105         }  
106  
107         if (typeof callbacks.onSuccess === 'function') {  
108             callbacks.onSuccess();  
109         }  
110     };  
111     > xhr.onerror = () => { ...  
115     };  
116     > xhr.onloadend = () => { ...  
120     };  
121     xhr.send();  
122 }
```

Никакие данные не возвращаются

Ничего не отправляем, `id` итак в URL'e



Улучшения

Конечно же, наше приложение можно улучшить, например:

1. Не перерисовывать каждый раз все элементы, а точно их удалять через API
2. Избавиться от `innerHTML`
3. Рисовать не общий `loader`, а на одну конкретную запись

И т.д.

Но наша с вами задача была отработать ключевые навыки, причём мы рассмотрели только один из сотен возможных вариантов построения API. Но уже отталкиваясь от этого, вы можете представлять себе, как всё устроено.



Синхронизация

На следующем курсе мы с вами будем рассматривать самую важную проблему: синхронизацию – что если пока вы добавляли новую запись, кто-то удалил одну из ваших старых? В текущей реализации вы этого не увидите, пока не обновите страницу.



innerHTML

Мы вам обещали рассказать о проблемах `innerHTML`. Так вот смотрите, попробуйте вбить в поле названия вот такое: `<button>click me</button>`

Вроде бы безобидно, но все, кто откроют нашу страницу получат:

- Название: `click me`, цена: 1000 с. `Удалить`
Some description

На самом деле, можно сделать не только такие безобидные вещи (и, например, вставить `onclick` – мы получили уязвимость, которая называется XSS, с помощью неё можно "красть" данные из браузера пользователя или выполнять другие нецелевые действия). Поэтому будьте аккуратны и старайтесь не использовать `innerHTML`.

Более подробно об этом мы будем говорить на специальном курсе, посвящённом веб-безопасности.



Удаление обработчиков

Стоит сказать пару слов об удалении обработчиков, установленных с помощью `addEventListener` (этот вопрос часто задают на собеседованиях). Поскольку функция – это объект, то удалить обработчик можно только в случае если его сохранить в какую-то переменную:

```
216   const handler = () => console.log('clicked');  
217   rootEl.addEventListener('click', handler);  
218   rootEl.removeEventListener('click', handler);
```

А вот так нельзя (потому что это два разных объекта, несмотря на то, что код в них вроде одинаковый):

```
216   rootEl.addEventListener('click', () => console.log('clicked'));  
217   rootEl.removeEventListener('click', () => console.log('clicked'));
```



Итоги



ИТОГИ

В этой лекции мы посмотрели с вами как работать с HTTP. На самом деле, это всего лишь первые шаги и нам ещё предстоит познакомиться с такими вещами как Promise, async/await, а также поговорить об обработке ошибок в JS коде.



Домашнее задание



Орг.моменты

Практикум состоит из 8 обязательных занятий. Мы выкладываем новые занятия каждый понедельник в 14:00 (по Душанбе), кроме первой недели.

Каждое воскресенье в 23:59 (по Душанбе) дедлайн сдачи домашнего задания. Дедлайн – это предельный срок, до которого вы должны сдать ДЗ.

Если не успеете сдать в срок домашнее задание, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

Все вопросы вы сможете задавать в [Телеграм канале](#).



Сервер

Сервер вы можете скачать по адресу <https://alif-skills.pro/media/homework-server.js>

Сервер один для всех ДЗ, но пути, по которым нужно делать запросы – различаются, будьте внимательны.

Сервер запускается командой `node server.js`. Если вдруг, при запуске вы видите ошибки, то обратитесь к первым слайдам этой лекции. Если это не поможет - пишите в канал курса.



Сервер

Важно: сервер не хранит данные, поэтому если вы его выключили или случайно "уронили", отправив неправильные данные, то всё, что в нём хранилось (все ваши добавленные объекты) сотрутся.

Сервер достаточно непривередливый и если вы отправляете "неправильные" данные, то он в большинстве случаев добавит их "как есть" и уже вам самим придётся разбираться с тем, что добавлено. Поэтому внимательно следите через панельку Network, что и куда вы отправляете (вы всегда можете перезапустить сервер, чтобы получить "чистое" состояние).



Постановка ДЗ

Поскольку это последняя лекция, ДЗ в ней сформулированы так, как достаточно часто формулируются в реальной жизни. Это не хорошо и не плохо – с этим просто нужно учиться работать.

Напоминаем, что если что-то будет непонятно, вы всегда можете спросить в канале курса.



ДЗ №1: Media Wall

Это повторение задачи №1 из 7-ой лекции, но теперь данные вам приходят с сервера, а не "зашиты" у вас локально.

Для того, чтобы получить данные нужно сделать GET-запрос на URL

<http://127.0.0.1:9999/api/hw29/posts>



ДЗ №1: Media Wall

Данные придут в формате JSON:

```
✓ [
  ✓ {
    "id": 3,
    "type": "text",
    "content": "Final Week!"
  },
  ✓ {
    "id": 2,
    "type": "image",
    "content": "img/logo_js.svg"
  },
  ✓ {
    "id": 1,
    "type": "video",
    "content": "video/video.mp4"
  }
]
```

Важно: бот будет подменять данные, поэтому не думайте, что вам придёт только три элемента и только таких. Но типы будут именно такие: [text](#), [image](#), [video](#).



ДЗ №1: Media Wall

Загрузка данных должна происходить сразу при загрузке страницы.

На время загрузки вы должны показывать loader:

```
<div data-id="loader">Данные загружаются</div>
```

После загрузки, loader должен удаляться со страницы (не скрываться, а именно удаляться из DOM).



ДЗ №1: Media Wall

Генерируемое DOM-дерево для постов должно быть вот таким:

```
<div data-type="text" data-id="3">
|   <div>Final Week!</div>
</div>
<div data-type="image" data-id="2">
|   
</div>
<div data-type="video" data-id="1">
|   <video src="video/video.mp4" controls></video>
</div>
```

Поместите все элементы внутрь `<div data-id="wall"></div>`

Все данные должны храниться внутри массива с именем `posts`.

Бот будет искать в архиве с решением каталог `media-wall`.



ДЗ №2: Retry

Достаточно часто при сетевых запросах случаются ошибки. Это может быть по разным причинам: неправильно написан сервер, плохое соединение и т.д.

Нам это не принципиально, а важно следующее: нужно давать возможность пользователю повторить запрос без полной перезагрузки страницы.

Что мы хотим сделать: по умолчанию пользователю должна загружаться вот такая страница с одной кнопкой:

Загрузить данные



ДЗ №2: Retry

Загрузить данные

Вы нажимаете на неё, отправляется GET-запрос на адрес <http://127.0.0.1:9999/api/hw30/posts> (не забудьте показать `loader`) и если с сервера пришёл код ответа 2xx (от 200 до 299), то вы отрисовываете посты так же, как в задании Media Wall. При этом кнопка "Загрузить данные" никуда не пропадает.

Если на неё нажать повторно, то снова будет отправлен запрос на тот же адрес, данные снова загрузятся, содержимое `<div data-id="wall"></div>` очистится и туда заново отрисуются ваши элементы.

У кнопки "Загрузить данные" должен быть установлен `data-action="load"`.



ДЗ №2: Retry

Если же вернётся код не 2xx, то кнопка "Загрузить данные" должна исчезнуть (удалиться из DOM), а на её месте должно появиться сообщение:

```
<div>
```

```
  Произшла ошибка: <span data-id="error">текст ошибки</span> <button  
data-action="retry">Повторить запрос</button>  
</div>
```

Откуда брать текст ошибки? Сервер при возникновении ошибки будет присылать вам вот такой JSON:

```
{  
  "message": "Сервер временно недоступен, попробуйте повторить ваш запрос позже"  
}
```

Отвечать таким JSON'ом сервер будет примерно на каждый 3-ий запрос.



ДЗ №2: Retry

Все данные должны храниться внутри массива с именем `posts`. При нажатии на кнопку "Повторить запрос" данные снова должны загружаться (а сам контейнер с ошибкой и кнопкой "Повторить запрос" - удаляться) и появляться кнопка "Загрузить данные" (так до тех пор, пока снова не придёт ошибка).

Бот будет искать в архиве с решением каталог `retry`.



ДЗ №3: Library

Если вы внимательно читали лекцию, то могли заметить, что функции, которые мы пишем, отличаются всего парой строк, а весь остальной код дублируется.

Дублирование кода – это всегда очень плохо. Когда вы только пишете код, оно допустимо, но затем нужно проводить рефакторинг – улучшать структуру кода без изменения его функциональности. Т.е. грубо говоря, "наводить чистоту" в коде. Если этого не делать, то код превратится в "помойку", с которой не возможно работать.



ДЗ №3: Library

Что вам нужно сделать: нужно написать универсальную функцию `ajax`, которая будет отправлять запросы всех типов.

Вот так должна выглядеть функция:

```
1  function ajax(method, url, headers, callbacks, body) {  
2  
3  }
```

Функция внутри должна только создавать объект XMLHttpRequest, выставить метод, заголовки и устанавливать callback'и:

- `onStart` – в самом начале
- `onFinish` – при окончании запроса
- `onError` – при событии типа error или не 2xx статус коде
- `onSuccess` – при событии типа `load`



ДЗ №3: Library

Функция не должна вызывать `JSON.stringify` или `JSON.parse` – готовые данные ей передаёт вызывающий код. Как это выглядит (примерно так будет вызывать бот):

```
// для тестирования успешного ответа
ajax('GET', 'http://127.0.0.1:9999/api/hw31/success', {}, {
  onStart: () => {/* показываем loader */},
  onFinish: () => {/* скрываем loader */},
  onError: error => {/* в error либо строка "Network Error" либо статус ответа (statusText) */},
  onSuccess: data => {/* в data.responseText */},
});
```

```
// для тестирования ошибки
ajax('GET', 'http://127.0.0.1:9999/api/hw31/error', {}, {
  onStart: () => {/* показываем loader */},
  onFinish: () => {/* скрываем loader */},
  onError: error => {/* в error либо строка "Network Error" либо статус ответа (statusText) */},
  onSuccess: data => {/* в data.responseText */},
});
```

```
// для тестирования заголовков и тела запроса/ответа
ajax('POST', 'http://127.0.0.1:9999/api/hw31/success', {'Content-Type': 'application/json'}, {
  onStart: () => {/* показываем loader */},
  onFinish: () => {/* скрываем loader */},
  onError: error => {/* в error либо строка "Network Error" либо статус ответа (statusText) */},
  onSuccess: data => {/* в data.responseText */},
}, JSON.stringify({message: 'test'}));
```



ДЗ №3: Library

Обратите внимание:

1. Вам придётся разобраться со замечательным объектом [Object](#) и его методами, чтобы научиться извлекать заголовки
2. В callback'ах не обязательно будут присутствовать все 4 свойства, ваш код должен корректно это обрабатывать
3. [body](#) должен отправляться

Вы можете сами протестировать эту функцию на тех URL'ах, что мы вам показали. Отрисовывать данные, [loader](#) и всё остальное не нужно – вы делаете библиотеку. Это значит, что ваш код будут подключать к другим проектам и использовать.

Бот будет искать в архиве с решением каталог [library](#).



ДЗ №4: Posts

А вот это уже полноценный сервис с возможностью получения списка постов с сервера, добавления и удаления. Вам предстоит сделать почти всё то же самое, что мы делали на лекции.

У вас есть URL: <http://127.0.0.1:9999/api/hw32/posts>, с которого с помощью GET-запроса можно получать список постов вида:

```
✓ [
  ✓ {
    "id": 2,
    "author": "JS Pro",
    "text": "JS is fun!"
  },
  ✓ {
    "id": 1,
    "author": "JS Pro",
    "text": "JS rulezzz!"
  }
]
```



ДЗ №4: Posts

Для добавления поста необходимо отправить POST-запрос на URL

<http://127.0.0.1:9999/api/hw32/posts> с заголовком "Content-Type": "application/json" и

телом в виде JSON:

```
{  
  "id": 0,  
  "author": "JS Pro",  
  "text": "JS rulezzz!"  
}
```

А ответ уже приходит с заполненным id:

```
{  
  "id": 1,  
  "author": "JS Pro",  
  "text": "JS rulezzz!"  
}
```



ДЗ №4: Posts

Для удаления поста необходимо отправить DELETE-запрос на URL

<http://127.0.0.1:9999/api/hw32/posts/{id}>, где id – идентификатор существующего поста, например: <http://127.0.0.1:9999/api/hw32/posts/1>



ДЗ №4: Posts

Разметка должна выглядеть следующим образом:

```
<div id="root">
  <div data-id="loader">Идёт загрузка</div>
  <form data-id="post-form">
    <fieldset data-id="post-fields">
      <input data-input="author">
      <input data-input="text">
      <button data-action="add">Добавить</button>
    </fieldset>
  </form>
  <div data-id="posts">
    <div data-type="post" data-post-id="2">
      <div data-post-part="author">JS PRO</div>
      <div data-post-part="text">JS is fun!</div>
      <div data-post-action="remove">Удалить</div>
    </div>
    <div data-type="post" data-post-id="1">
      <div data-post-part="author">JS PRO</div>
      <div data-post-part="text">JS rulezzz!</div>
      <div data-post-action="remove">Удалить</div>
    </div>
  </div>
</div>
```

мы не ошиблись, тут `div`
на нём тоже можно кликать



ДЗ №4: Posts

Соответственно:

1. При загрузке страницы – загрузка данных посредством HTTP-запроса (не забывайте про **loader** – для разнообразия в этот раз будем его просто скрывать через **display** или **visibility**)
2. При нажатии на кнопку "**Добавить**" – добавление с HTTP-запросом и всеми проверками (наверное, где-то нужно сообщение, если одно из полей пустое)
3. При нажатии на кнопку "**Удалить**" – удаление с сервера и из DOM (из DOM можете удалять сразу, не дожидаясь прихода ответа с сервера).

Все посты должны храниться в переменной с именем **posts**.

Бот будет искать в архиве с решением каталог **posts**.



ДЗ №5: Critic

Промоделируем ситуацию, при которой тот проект, что вы делали, начинает развиваться и обрести новую функциональность.

Первое, что мы добавим в проект из предыдущего ДЗ – это возможность ставить лайки и дизлайки соответственно. Что изменится? Во-первых, теперь все URL'ы будут не hw32, а hw33. А во-вторых, во все модели постов (то, как мы описываем пост в виде объекта достаточно часто называют моделью – совокупность свойств: `id`, `author` и `text`) будут содержать свойство `likes`.



ДЗ №5: Critic

При добавлении **likes** должно быть равно 0:

```
{  
  "id": 0,  
  "author": "JS Pro",  
  "text": "JS rulezzz!",  
  "likes": 0  
}
```



ДЗ №5: Critic

При получении постов мы тоже будем получать это свойство:

```
✓ [
  ✓ {
    "id": 2,
    "author": "JS Pro",
    "text": "JS is fun!",
    "likes": 0
  },
  ✓ {
    "id": 1,
    "author": "JS Pro",
    "text": "JS rulezzz!",
    "likes": 1
  }
]
```

Удаление поста остаётся без изменений. А вот самое интересное, это как ставить лайки и дизлайки.



ДЗ №5: Critic

Для установки лайка нужно отправить POST-запрос на URL (никаких заголовков и тела не нужно): <http://127.0.0.1:9999/api/hw33/posts/1/likes>



id поста

Для установки дизлайка (или -1) нужно отправить DELETE-запрос на URL (никаких заголовков и тела не нужно): <http://127.0.0.1:9999/api/hw33/posts/1/likes>



id поста



Разметка:

```
<div id="root">
  <div data-id="loader">Идёт загрузка</div>
  <form data-id="post-form">
    <fieldset data-id="post-fields">
      <input data-input="author">
      <input data-input="text">
      <button data-action="add">Добавить</button>
    </fieldset>
  </form>
  <div data-id="posts">
    <div data-type="post" data-post-id="2">
      <div data-post-part="author">JS PRO</div>
      <div data-post-part="text">JS is fun!</div>
      <div>
        ♥ <span data-info="likes">0</span>
        <button data-action="like">+1</button>
        <button data-action="dislike">-1</button>
        <button data-action="remove">Удалить</button>
      </div>
    </div>
    <div data-type="post" data-post-id="1">
      <div data-post-part="author">JS PRO</div>
      <div data-post-part="text">JS rulezzz!</div>
      <div>
        ♥ <span data-info="likes">1</span>
        <button data-action="like">+1</button>
        <button data-action="dislike">-1</button>
        <button data-action="remove">Удалить</button>
      </div>
    </div>
  </div>
</div>
```



ДЗ №5: Critic

А теперь самое важное: поскольку запросы идут долго, мы хотим, чтобы пока идёт запрос, все кнопки скрывались и показывался ладер (но не общий, а в каждом конкретном посте – т.е. пользователь кликнул на лайк и вместо кнопок показывается ладер):

JS PRO

JS is fun!



JS PRO

JS rulezzz!



1

+1

-1

Удалить

В коде:

```
<div data-type="post" data-post-id="2">  
  <div data-post-part="author">JS PRO</div>  
  <div data-post-part="text">JS is fun!</div>  
  <div>  
    <span data-id="action-loader"></span>  
  </div>  
</div>
```



ДЗ №5: Critic

Про то, что сервер может ответить ошибкой, можете пока не думать.

Изменять количество лайков нужно только после прихода ответа.

Бот будет искать в архиве с решением каталог [critic](#).



ДЗ №6: Discussions

Продолжаем улучшать наш проект. Мы решили, что неплохо бы к каждому посту добавить возможность оставлять комментарии:

JS PRO

JS is fun!



0

+1

-1

Удалить

Добавить

Первый комментарий

Второй комментарий



ДЗ №6: Discussions

Разметка:

```
<div data-type="post" data-post-id="2">
  <div data-post-part="author">JS PRO</div>
  <div data-post-part="text">JS is fun!</div>
  <div>
    ♥ <span data-info="likes">0</span>
    <button data-action="like">+1</button>
    <button data-action="dislike">-1</button>
    <button data-action="remove">Удалить</button>
  </div>
  <form data-form="comment">
    <input data-id="text">
    <button>Добавить</button>
  </form>
  <div data-post-part="comments">
    <div data-comment-id="1">Первый комментарий</div>
    <div data-comment-id="2">Второй комментарий</div>
  </div>
</div>
```



ДЗ №6: Discussions

Что меняется? Все запросы теперь на hw34, а не hw33.

Кроме того, в постах сразу будут приходить комментарии:

```

  [
    {
      "id": 2,
      "author": "JS Pro",
      "text": "JS is fun!",
      "likes": 0,
      "comments": []
    },
    {
      "id": 1,
      "author": "JS Pro",
      "text": "JS rulezzz!",
      "likes": 0,
      "comments": [
        {
          "id": 1,
          "text": "Первый комментарий"
        },
        {
          "id": 2,
          "text": "Второй комментарий"
        }
      ]
    }
  ]

```



ДЗ №6: Discussions

При создании поста нужно передавать свойство `comments` со значением, равным пустому массиву:

```
{  
  "id": 0,  
  "author": "JS Pro",  
  "text": "JS rulezzz!",  
  "likes": 0,  
  "comments": []  
}
```



ДЗ №6: Discussions

Для создания комментария нужно отправить POST-запрос с заголовком Content-Type: application/json и телом на адрес:

<http://127.0.0.1:9999/api/hw34/posts/1/comments>

```
{  
  "id": 0,  
  "text": "Первый комментарий"  
}
```

id поста

В ответе будет возвращаться проставленный id (после этого можно добавлять элемент в DOM):

```
✓ {  
  "id": 1,  
  "text": "Первый комментарий"  
}
```



ДЗ №6: Discussions

Ну и как вы уже, наверное, догадались, мы хотим, чтобы на время отправки запроса вместо формочки показывался ладер (сами комментарии скрывать не нужно):

JS PRO

JS is fun!



Первый комментарий

Второй комментарий

Обратите внимание: механика показа и скрытия такая же, как в предыдущем ДЗ. Форма удаляется из DOM, вместо неё подставляется loader, затем loader удаляется, показывается форма. У loader'а, как и в предыдущем ДЗ:

```
<span data-id="action-loader"></span>
```



ДЗ №6: Discussions

Бот будет искать в архиве с решением каталог [discussions](#).



ДЗ №7: Post Editor

Продолжаем улучшать наш проект. Мы решили, что неплохо бы дать возможность пользователю редактировать уже созданные посты:

```
<div data-type="post" data-post-id="2">
  <div data-post-part="author">JS PRO</div>
  <div data-post-part="text">JS is fun!</div>
  <div>
    ♥ <span data-info="likes">0</span>
    <button data-action="like">+1</button>
    <button data-action="dislike">-1</button>
    <button data-action="edit">Изменить</button>
    <button data-action="remove">Удалить</button>
  </div>
  <form data-form="comment">
    <input data-id="text">
    <button>Добавить</button>
  </form>
  <div data-post-part="comments">
    <div data-comment-id="1">Первый комментарий</div>
    <div data-comment-id="2">Второй комментарий</div>
  </div>
</div>
```



ДЗ №7: Post Editor

Что происходит по нажатию на кнопке? Заполняется исходная форма:

JS PRO	JS is fun!	Сохранить	Отмена
--------	------------	-----------	--------

При этом вместо кнопки "Добавить" появляются "Сохранить" и "Отмена".

Как заполняются поля? В свойство `value` можно писать и это приведёт к заполнению поля.

Что делать с `id`? `id` можно хранить в специальном `input`'е с типом `hidden` (не отображаемый). По умолчанию у него `value="0"` (вспоминайте, что мы при добавлении всегда отправляли `id=0`):

```
<form data-id="post-form">
  <fieldset data-id="post-fields">
    <input data-input="id" type="hidden" value="0">
    <input data-input="author">
    <input data-input="text">
    <button data-action="save">Сохранить</button>
    <button data-action="cancel">Отмена</button>
  </fieldset>
</form>
```



ДЗ №7: Post Editor

При нажатии на кнопку "Отмена" – всё возвращается к первоначальному состоянию.

При нажатии на кнопку "Сохранить" отправляется такой же запрос, что и на добавление, только в качестве id указывается id существующего поста. После получения ответа форма должна вернуться к первоначальному виду, а сам пост "обновится" (запрашивать заново все посты с сервера не нужно).

Не забывайте, что для этого ДЗ в URL'e у вас будет уже hw35.

Бот будет искать в архиве с решением каталог [editor](#).



ДЗ №8: Wall Auto Updater

И финальное улучшение. В чём его суть: мы хотим получать новые посты без перезагрузки страницы. О чём идёт речь? Представьте вы загрузили ленту – у вас загрузилось два поста. В этот момент ваш коллега, написал ещё пару постов. Но у вас этих постов нет, по одной простой причине – вы больше не ходили на сервер (не делали запрос по загрузке данных).

Самое время это сделать. Для вас сделали API, которое позволяет вам получать посты по id новее определённого. Что это значит? Это значит, что если вы загрузили два поста – то у первого id = 1, у второго = 2. Вам не нужно их заново грузить. Вы хотите получить все посты новее 2.

Для этого вам нужно сделать вот такой GET-запрос:

<http://127.0.0.1:9999/api/hw36/posts/newer/2> ← id поста



ДЗ №8: Wall Auto Updater

Установите выполнение этого запроса через каждые 5 секунд с помощью `setInterval` и добавляйте новые посты до старых. Это не самый лучший способ (можете посмотреть, как это делают соц.сети – они не добавляют посты, а отображают кнопку "появились новые посты"), потому что если вы постоянно будете добавлять, а человек в это время читает, то посты будут "съезжать" вниз. Но мы пока об этом не думаем.

Сервер вам будет возвращать либо посты новее, указанного вами, либо пустой массив, если новее нет.

Как себя проверить? Открываете две вкладки: в одной добавляете и смотрите на вторую – то, что вы добавили в первой, должно через 5 секунд автоматически загрузиться во вторую. Никаких лоадеров и прочего для "подгрузки" рисовать не нужно.



ДЗ №8: Wall Auto Updater

Бот будет искать в архиве с решением каталог [auto-wall](#).



Спасибо за внимание

alif skills

2023г.

